Uniwersytet im. A. Mickiewicza w Poznaniu

Wydział Matematyki i Informatyki

Praca magisterska

Faktoryzowanie danych wyjściowych w neuronowym tłumaczeniu automatycznym

Factoring the Output of the Neural Machine Translation

Marek Szawioła

Numer albumu: 375367

Kierunek: Informatyka, Specjalność: Systemy inteligentne

Promotor:

prof. UAM dr hab. Krzysztof Jassem

Poznań, 2019

Poznań, dnia 30 września 2019r.

Oświadczenie

Ja, niżej podpisany/a Marek Szawioła, student wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt.
Faktoryzowanie danych wyjściowych w neuronowym tłumaczeniu automatycznym napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej.

Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania administracyjnego w sprawie nadania tytułu zawodowego.

[ ]* - wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[ ]* - wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

*Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni Archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnianie pracy.

.....

(czytelny podpis studenta)

Spis treści

Wstęp **9**

1. Podstawy teoretyczne **11**

1.1. Rys historyczny 11

1.1.1. Początki 11

1.1.2. Czasy współczesne 12

1.2. Ewaluacja 14

1.2.1. Metryka Precision 14

1.2.2. Metryka BLEU 15

2. Narzędzia **18**

2.1. Marian 18

2.1.1. Podstawowe informacje 18

2.1.2. Struktura danych wejściowych 20

2.1.3. Wyjście 22

2.2. Tree-tagger 22

2.2.1. Podstawowe informacje 23

2.2.2. Struktura danych wejściowych 23

2.2.3. Wyjście 24

2.2.4. TreeTagger Python Wrapper 24

3. Przetwarzanie wstępne **29**

3.1. Początkowe operacje 29

3.1.1. Gromadzenie danych tekstowych 29

3.1.2. Skracanie korpusów 31

3.1.3. Czyszczenie korpusów (ang. *cleaning*) 32

3.2. Tokenizacja 33

3.2.1. Implementacja w projekcie 35

3.3. Normalizacja słów oraz algorytm BPE 40

3.3.1. Truecasing 40
3.3.2. Byte Pair Encoding 42

4. Tagging **45**
4.1. Oznaczanie części mowy 45
4.1.1. Idea tagowania 47
4.2. Zestaw tagów Penn Treebank 49
4.2.1. Objaśnienie tagów 51
4.3. Proces tagowania 57

5. Projekt magisterski **59**
5.1. Gromadzenie i wstępne przetwarzanie 59
5.1.1. Gromadzenie danych tekstowych 59
5.1.2. Czyszczenie korpusów 60
5.2. Tagowanie i tokenizacja 61
5.2.1. Implementacja tagowania w projekcie 61
5.3. Ostatnie szlify 66
5.3.1. Rozdzielanie tekstu i tagów 67
5.3.2. Wstawianie tagów 69
5.4. Trening modelu tłumacza 70
5.5. Tłumaczenie i postprocessing 74

6. Ewaluacja modeli i wnioski **78**
6.1. Pomiar jakości 78
6.2. Konkluzje 80

A. Kod źródłowy tokenizacji tekstów niemieckich **82**

B. Kod źródłowy tagowania tekstów angielskich **86**

Streszczenie

Praca weryfikuje wpływ faktoryzowania danych wyjściowych, w neuronowym tłumaczeniu automatycznym, na jakość tłumaczenia. Przeprowadzony eksperyment obejmował konstrukcję automatycznego tłumacza uwzględniającego znaczniki reprezentujące części mowy po stronie docelowej danych treningowych. Zarówno etykiety jak i tekst umieszczono we wspólnym pliku tekstowym. Opracowanie prezentuje proces przygotowania danych treningowych, konstrukcję tłumacza oraz procedurę ewaluacji wyników. Ponadto scharakteryzowano narzędzia informatyczne używane w badaniach oraz implementację stosownych operacji , w szczególności proces tagowania zbiorów języka docelowego, kluczowy dla trenowania weryfikowanego tłumacza. Finalnie dokonano oceny rezultatów uzyskanych w wyniku tłumaczenia wytrenowanymi modelami.

Słowa kluczowe

tłumaczenie automatyczne, faktoryzacja danych wyjściowych, tagowanie, POS-tagi, znaczniki, tokenizacja, narzędzie Marian, narzędzie TreeTagger

Abstract

This thesis verifies the impact of factorization of output data, in neural machine translation, on the quality of translation. The experiment involved the construction of a machine translator including tags representing parts of speech on the target side of training data. Both labels and text were placed in a common text file. The work describes the process of preparing training data, the construction of the translator and the procedure for evaluating the results. In addition, IT tools used in research and the implementation of relevant operations were characterized, in particular the process of tagging target language sets, crucial for training a verified translator. Finally, the results obtained as a result of translation with trained models were evaluated.

Key words

machine translation, output factorization, tagging, POS-tagging, tags, tokenization, Marian tool, TreeTagger tool

Wstęp

Efektywne tłumaczenie tekstu w dobie cyfrowej determinuje skuteczną międzynarodową komunikację. Warunkuje powszechny dostęp do szeroko rozumianej informacji. W ten kontekst wpisuje się przedłożone opracowanie, w którym opisano jedną z prób ulepszenia jakości tłumaczenia z odwołaniem do pakietu narzędzi Marian [14].

W niniejszej pracy starano się uzyskać odpowiedź na pytanie o wpływ faktoryzowania danych wyjściowych w neuronowym tłumaczeniu automatycznym. Dla jej potrzeb postawiono roboczą hipotezę, że dodanie znaczników reprezentujących części mowy do danych treningowych w języku docelowym poprawia jakość wyników, przy wykonaniu ewaluacji zbiorów testowych metodą BLEU.

Większość wytrenowanych dotychczas tłumaczy stosowało całkowicie inne podejście. Jeszcze niedawno tłumacze automatyczne były konstruowane na podstawie podejścia statystycznego (np. narzędzie Moses (`http://www.statmt.org/moses/`), które opisał Philipp Koehn w pozycji *Statistical Machine Translation*[15]. Z czasem, wraz z rozwojem technik sieci neuronowych, bardziej obiecujące okazało się konstruowanie tak zwanych neuronowych tłumaczy automatycznych (opisane na przykład w pracy [9]). Pośród różnych metod budowania takich tłumaczy, pojawiły się sposoby wykorzystujące w trenowaniu tłumacza tagi cech językowych. Jedną z nich stanowiło podejście odnoszące się do faktorów (ang. *factors*) języka źródłowego (praca [25]). Pojawił się również pomysł wykorzystania w treningu POS-Tagów języka docelowego, którą to ideę opisano w pracy [18]. Istotą jej jest modyfikacja algorytmu *encoder-decoder* tak, by uwzględniał on dodatkowe informacje o słowach, przekazywane do treningu.

Niniejsza praca proponuje połączenie znaczników ze zbiorem tekstowym według schematu: każdy tag następuje po odpowiadającym mu słowie. Zarówno etykiety jak i tekst powinny być zawarte we wspólnym pliku tekstowym.

Opracowanie prezentuje poszczególne etapy zrealizowane w celu właściwego przygotowania danych, konstrukcji tłumacza oraz przeprowadzenia ewaluacji wyników otrzymanych

z jego wykorzystaniem.

W rozdziale 1. zamieszczono podstawy teoretyczne syntetycznie wyjaśniające w jakim miejscu, z historycznego punktu widzenia, znajduje się obecnie tłumaczenie automatyczne (1.1). Przedstawiono metodę ewaluacji jaką zastosowano w ocenie wyniku przetwarzania (1.2).

Narzędzia informatyczne, wykorzystywane w badaniach, scharakteryzowano w rozdziale 2. Zaprezentowano tam wspomniany pakiet Marian, z którego użyciem przeprowadzono trening tłumacza (2.1) a także narzędzie TreeTagger (2.2) i odpowiadający mu tzw. wrapper języka Python wykorzystany do tagowania zbioru treningowego.

Dalej następuje przedstawienie operacji, przeprowadzonych na wszystkich zbiorach używanych podczas realizacji projektu, wraz z opisem ich implementacji (rozdział 3). Poruszono tutaj wątek początkowych działań (3.1), takich jak gromadzenie danych, przycinanie korpusów do potrzebnych rozmiarów oraz ich czyszczenie. Opisano też tokenizację (3.2) wraz z jej implementacją w związku z tekstami w języku źródłowym.

Kolejny rozdział wyjaśnia szczegóły procesu tagowania (rozdział 4). Zawarto tutaj jego zarys teoretyczny (4.1), określono stosowany zbiór tagów (4.2) oraz opisano samego procesu (4.3).

Zasadnicze działania programistyczne, wykonane w ramach realizacji projektu, zostały zebrane w rozdziale 5. Przypomniano detale wstępnych procesów realizowanych w ramach etapów przygotowania zbiorów tekstowych (5.1). W podrozdziale 5.2 przedstawiono szczegółowo implementację tagowania zbiorów języka docelowego, kluczowego dla trenowania weryfikowanego tłumacza. Również skomentowano finalne czynnościami, jakie musiały zostać wykonane w na zbiorach w związku z tagowaniem 5.3). Opisano implementacją treningu modelu tłumacza automatycznego (5.4) i samego tłumaczenia (5.5).

Całość projektu podsumowuje końcowy rozdział (6)), w którym zawarto ewaluację rezultatów uzyskanych w wyniku tłumaczenia wytrenowanymi modelami. Skonfrontowano te wyniki z pierwotną, roboczą hipotezą i sformułowano wniosek.

Rozdział 1

Podstawy teoretyczne

1.1. Rys historyczny

Na przestrzeni lat zmieniało się podejście do kwestii tłumaczenia. Początkowo zajmowały się tym osoby znające język, z którego następowało tłumaczenie (język źródłowy) i język na który tłumaczono (język docelowy). Wraz z rozwojem technik obliczeniowych i elektroniki, a w efekcie narzędzi informatycznych, zaczęto stosować translatory automatyczne. Poniżej przedstawiono ogólny rozwój podejścia do automatycznej translacji (rozdział 1.1.1) i przykładowe translatory dostępne w czasach współczesnych (rozdział 1.1.2).

1.1.1. Początki

Niniejszy rozdział opracowano na podstawie Koehn P.[16]

Początki tłumaczenia maszynowego sięgają ponad 60 lat wstecz. Już w pierwszej połowie XX w próbowano zaprząc maszyny do deszyfrowania wiadomości. W 1932r. grupa polskich kryptologów, głównie Marian Rejewski, Jerzy Różycki i Henryk Zygalski, złamała pierwszy szyfrogram wygenerowany przez Enigmę (niemiecką maszynę szyfrującą [13]). Podczas II Wojny Światowej dało to podwaliny dla wykorzystania pierwszych komputerów do dekodowania niemieckich szyfrów generowanych przez coraz nowocześniejsze maszyny Enigma. W 1947r. Warren Weaver stwierdził, że tłumaczenie przypomina deszyfrowanie wiadomości napisanej w znanym języku ale zakodowanej w nieoczywisty sposób.

Rozwój komputerów wybudził wysokie oczekiwania i wielkie nadzieje związane z tłumaczeniem maszynowym. Z coraz większą uwagą zaczęto badać ten obszar zastosowań

informatyki. Powstały nowe metody czy algorytmy służące już nie tylko rozszyfrowywaniu obcych języków. Sformułowane zasady, zależności znalazły zastosowanie w różnych gałęziach życia prywatnego i społecznego. Także rządy wielu krajów zwróciły uwagę na możliwość wykorzystania przetwarzania języka w systemach bezpieczeństwa (szczególnie w przypadku języków potencjalnie wrogich krajów).

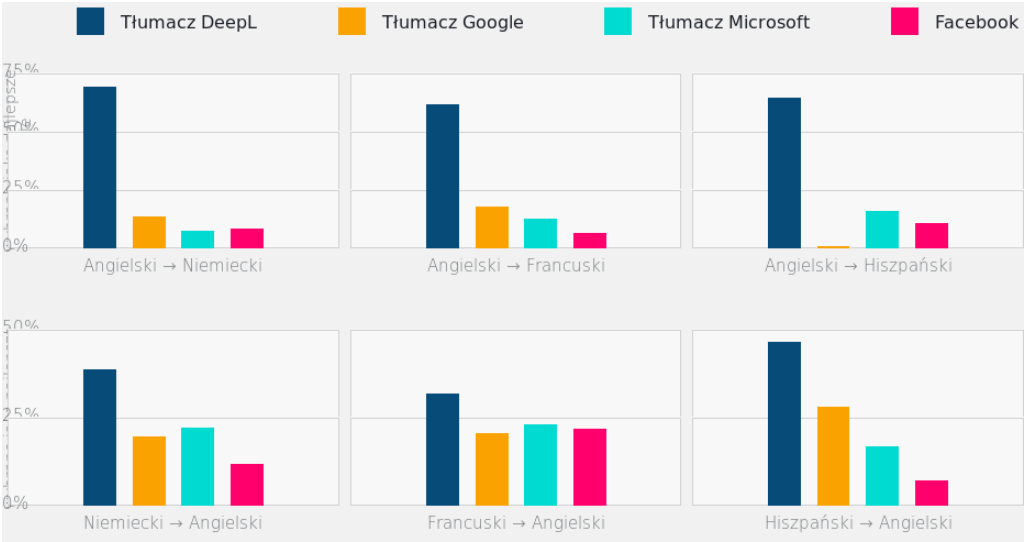
1.1.2. Czasy współczesne

Niniejszy rozdział opracowano na podstawie Och F.[20], Turovsky B.[27], DeepL Pro – FAQ[1], DeepL – Press[5]

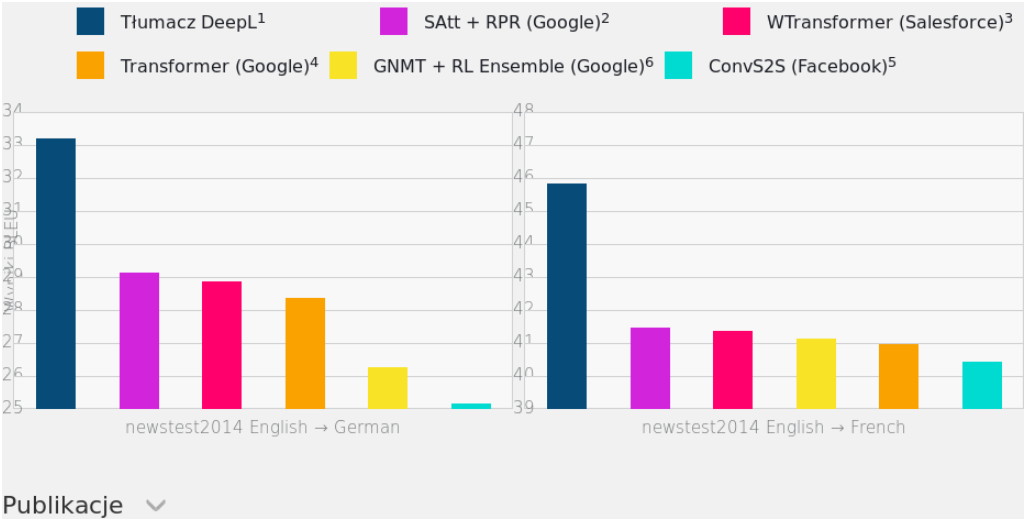
W obecnych czasach szczególnie dynamicznie rozwija się rynek tłumaczy maszynowych dostępnych online. Do jednych z najpopularniejszych z pewnością można zaliczyć usługę Tłumacz Google (ang. *Google Translate*) realizowaną przez Google LLC. Powstała ona w 2006 roku oferując możliwości tłumaczenia osiąganego metodą statystycznego tłumaczenia maszynowego ([20]). We wrześniu 2016 roku nastąpił znaczny krok w rozwoju narzędzia — wprowadzono Neuronowe Tłumaczenie Automatyczne (ang. *Neural Machine Translation*) ([27]). Od tego momentu podczas wykonywania tłumaczenia narzędzie nie koncentrowało się na małych fragmentach tłumaczonego tekstu (ang. *piece by piece*), ale zaczęło brać pod uwagę całe zdania. Uwzględnienie szerszego kontekstu pozwala osiągać bardziej trafne (ang. *relevant*) tłumaczenia, które następnie są poprawiane w celu osiągnięcia większej płynności (tekst wyjściowy jest bardziej zbliżony do poprawnej gramatycznie naturalnej wypowiedzi ludzkiej). W efekcie użytkownikowi dużo łatwiej zrozumieć tłumaczenia potrzebnych zdań, akapitów a nawet całych artykułów. Co więcej, Tłumacz Google rozwija się w miarę upływu czasu dzięki zastosowaniu systemu *end-to-end* zbudowanego na sieci neuronowej. Początkowo Tłumacz Google wzmocniony możliwościami sieci neuronowych obsługiwał tylko kilka najpopularniejszych języków (jak na przykład angielski, niemiecki, chiński, japoński czy turecki). W miarę upływu czasu dodano oczywiście kolejne (jak w marcu 2017 roku, kiedy udostępniono neuronowe tłumaczenie m.in. dla języka polskiego).

Konkurencja dla Tłumacza Google również wzrasta w siłę dzięki zastosowaniu technik sieci neuronowych. Przykładem może być Tłumacz DeepL opierający algorytm tłumaczenia na sztucznej inteligencji. Powstała w kwietniu 2017 roku firma DeepL (wcześniej Linguee) specjalizuje się w uczeniu głębokim (ang. *deep learning*) i rozwoju sztucznej inteligencji właśnie pod kątem tłumaczenia automatycznego. Według danych dostępnych na stronie `https://www.deepl.com/press.html`[5] tłumaczenia 100 zdań z i na język an-

gielski wyznaczane przez DeepL (dla języków niemieckiego, francuskiego i hiszpańskiego) bardziej były poprawne niż wyniki Tłumacza Google, Tłumacza Microsoft czy Facebook — oceny translatorów dokonywali tłumacze – osoby (testy przeprowadzono w 2017 roku). Według pomiarów przeprowadzonych w 2014 roku tłumacz DeepL okazał się najlepszy również według oceny BLEU (rys.1.1 i 1.2).



Rys. 1.1. Oceny tłumaczeń dokonane przez ludzi – wartości procentowe oznaczają dla jakiej części oceniających dany translator został uznany za najlepszy



Rys. 1.2. Ocena BLEU

Wykresy na Rys. 1.1 i Rys. 1.2 pochodzą ze strony `https://www.deepl.com/press.html`

1.2. Ewaluacja

W celu porównania jakości tłumaczy automatycznych można przede wszystkim zestawić jakości tłumaczeń określonego tekstu uzyskanych przy użyciu porównywanych translatorów. W tym miejscu pojawia się problem ilościowej ewaluacji tłumaczeń. Jak zmierzyć , które tłumaczenie jest lepsze?

Można posłużyć się ludzką oceną tłumaczenia. Taka metoda wiąże się jednak z obciążeniami, takimi jak wymagania dużej ilości czasu czy kwestia nakładów finansowych (oceniającym zwykle trzeba zapłacić).

Wobec potrzeby szybkiej i taniej ewaluacji najwygodniejszy jest odpowiedni algorytm, program wyliczający potrzebną metrykę. W podrozdziałach przedstawiona została podstawowa metryka *precision* (rozdział 1.2.1) i wywodząca się z niej popularna metryka *BLEU* (rozdział 1.2.2).

1.2.1. Metryka Precision

REFER: the programmer must send changes to the repository
TLUM1: the developer must submit changes to the reposotory
TLUM2: repository must receive changes from the programmer

Rys. 1.3. Precision — przykład

Prostą metodą porównania tłumaczeń jest znalezienie wspólnych słów między tłumaczeniem weryfikowanym a zbiorem referencyjnym. By uzyskać konkretną wartość zliczamy liczbę słów wspólnych dla tłumaczenia i zbioru referencyjnego. Nazywamy je słowami poprawnymi. Przykładowe zdania zostały zamieszczone na rysunku 1.3. Stosunek liczby słów poprawnych do liczby wszystkich słów w tłumaczeniu daje miarę *precision* zdefiniowaną równaniem (1.1).

$$precision = \frac{correct}{output - length} \tag{1.1}$$

Dla podanego przykładu 1.3 miara *precision* przyjmuje wartości zebrane w tabeli 1.1

Tab. 1.1. Ewaluacja tłumaczeń

Metryka	**ZD1**	**ZD2**
precision	6/8	5/7

Biorąc pod uwagę rozważaną metrykę (1.1) można wnioskować, że zdanie wyjściowe *TLUM1* lepiej oddaje znaczenie zdania referencyjnego, innymi słowy tłumaczenie *TLUM1* jest bardziej poprawne według metryki *precision*.

1.2.2. Metryka BLEU

Najpopularniejszą metryką używaną w przypadku ewaluacji tłumaczeń jest metryka BLEU (ang. Bilingual Evaluation Understudy). W przeciwieństwie do Precision korzysta się w niej nie tylko z liczby pojedynczych słów poprawnych (jak to zachodzi w przypadku Precision), pod uwagę brane są większe n-gramy. Za pomocą n-gramów (np. 1-gram, 2-gram, 3-gram, ...) określa się jak duże grupy składające się z sąsiadujących słów, musi być wspólna między tłumaczeniem a zbiorem referencyjnym. W przypadku metryki precision rozważano tylko grupy jednowyrazowe — 1-gramy.

REFER: the programmer must send changes to the repository
TLUM1: the developer must submit changes to the reposotory
TLUM2: repository must receive changes from the programmer

Rys. 1.4. BLEU — przykład

Rysunek 1.4 ilustruje najdłuższe poprawne n-gramy na przykładzie zdań używanych w przypadku metryki *precision*. Dla *TLUM1* wytępuje jeden 4-gram changes to the repository oraz dwa 1-gramy the i must. Z kolei dla *TLUM2* występuje jeden 2-gram the programmer oraz trzy 1-gramy repository, must i changes.

Dysponując liczbą n-gramów można obliczyć stosunek liczby poprawnych n-gramów w relacji do liczby wszystkich n-gramów występujących w tłumaczeniu (dla danej wartości n). W przykładowych zdaniach, dla n przyjmującego wartości od 1 do 4, stosunki liczby poprawnych n-gramów do liczby wszystkich n-gramów danego rzędu wynoszą:

- TLUM1: 1-gram - 6/8, 2-gram - 3/7, 3-gram - 2/6, 4-gram - 1/5
- TLUM2: 1-gram - 5/7, 2-gram - 1/6, 3-gram - 0/5, 4-gram - 0/4

Na podstawie obliczonych stosunków oraz poniższego wzoru można wyliczyć metrykę BLEU definiowaną jako (1.2).

$$BLEU - n = brevity - penalty \cdot \exp \sum_{k=1}^{n} \lambda_i \log precision_i \tag{1.2}$$

W metrykach opierających się o miarę *precision* występuje problem ze zbyt krótkimi zdaniami w tekście tłumaczenia. W równaniu nie występuje żaden element uwzględniający pominięcie w tłumaczeniu niektórych słów. Wyliczając *BLEU* rozwiązuje się ten problem korzystając z dodatkowego czynnika *brevity-penalty* (1.3).

$$brevity - penalty = \min\left(1, \frac{output - length}{reference - length}\right) \tag{1.3}$$

W przypadku zbyt krótkich danych tekstowych pomniejsza on obliczaną wartość miary. Dla przykładu z rysunku 1.4 wartości *brevity-penalty* wynoszą:

ZD1	**ZD2**
1	7/8

Jako maksymalną wartość n dla szukanych n-gramów zazwyczaj przyjmuje się 4. Co więcej również wartość wagi λ_i dla kolejnych wartości i najczęściej jest taka sama i wynosi 1. Przyjęcie takich wartości upraszcza formułę *BLEU* do postaci (1.4):

$$BLEU - 4 = \min\left(1, \frac{output - length}{reference - length}\right) \prod_{i=1}^{4} precision_i \tag{1.4}$$

Na podstawie zebranych danych możemy wyliczyć metrykę dla obu rozważanych zdań wyjściowych (tabela 1.2)

Tab. 1.2. Ewaluacja tłumaczeń

Metryka	**ZD1**	**ZD2**
precision (1-gram)	6/8	5/7
precision (2-gram)	3/7	1/6
precision (3-gram)	2/6	0/5
precision (4-gram)	1/5	0/4
BLEU-1	75%	63%
BLEU-2	32%	10%
BLEU-3	11%	0%
BLEU-4	2%	0%

Na uwagę zasługuje fakt, że pomimo poprawienia wiarygodności metryki *BLEU* przez uwzględnienie w niej negatywnego wpływu zbyt krótkich zdań, wciąż występuje problem z osiąganiem wartości 0 przez metrykę. Dla każdej sytuacji, gdy dla danego n-gramu któregokolwiek rzędu nie występuje dopasowanie między zdaniem referencyjnym a zdaniem wyjściowym (liczymy dla niego *BLEU*) metryka przyjmuje wartość 0. O ile często znajdu-

ją się dopasowania 1-gramów czy 2-gramów, o tyle przy wyliczaniu metryki dla pojedynczych zdań często nie ma dopasowań 4-gramów. W związku z tym metrykę *BLEU* często się wylicza na zbiorze wszystkich zdań wyjściowych. Dla większego zbioru wyjściowego wystąpienie danego n-gramu wysokiego rzędu (np. 4-gram) jest bardziej prawdopodobne.

Rozdział 2

Narzędzia

W rozdziale zastaną przedstawione narzędzia użyte w realizacji projektu magisterskiego. Każdy z podrozdziałów przedstawia inny framework używany przy wykonaniu zadań programistycznych. Pakiet Marian opisany został w rozdziale 2.1 a w rozdziale 2.2 przybliżono narzędzie TreeTagger wraz z udostępniającą jego możliwości biblioteką TreeTagger Wrapper (2.2.4).

2.1. Marian

Niniejszy rozdział opracowano na podstawie Junczys-Dowmunt M. i.in.[14], Koehn P. [17].

Pakiet narzędzi Marian rozwijano gównie na Uniwersytecie im. Adama Mickiewicza w Poznaniu i w placówce University of Edinburgh. Jest to wydajny framework służący do wykonywania zadań związanych z neuronowym tłumaczeniem maszynowym. Poniższe podrozdziały prezentują podstawowe informacje oraz przybliżają strukturę danych wejściowych stosowanych z narzędziem.

2.1.1. Podstawowe informacje

Framework Marian został stworzony w oparciu o zestaw narzędzi Nematus [24] służący realizacji zadań związanych z neuronowym tłumaczeniem maszynowym. Zachowano w nim kompatybilność z modelami utorzonymi narzędziem Nematus. Spośród cech charakterystycznych dla narzędzia Marian można wymienić kilka zamieszczonych poniżej.

- Po pierwsze Marian jest napisany z użyciem wyłącznie wersji 11 języka C++.

- Z założenia nie prezentuje żadnych powiązań z językiem Python: zarówno kod tworzenia modeli jak i meta-algorytmy zostały zaimplementowane w całości w języku C++ 11

- Marian jest narzędziem samodzielnym, samowystarczalnym. Dysponuje własnym, niezależnym back-endem.

Ponafto Marian, jako samodzielne oprogramowanie, odwołuje się do niewielkiej liczby zewnętrznych narzędzi. Korzysta jedynie ze wsparcia dostarczanego przez Boost, system CUDA lub bibliotekę BLAS. Programista otrzymuje tym samym możliwość niemal nieograniczonego rozwijania i optymalizacji kodu pakietu Marian na wielu poziomach. Obejmują one kwestie takie jak na przykład niestandardowe (własne) jądra GPU (ang. *GPU kernels*), implementacja meta-algorytmów (ang. *meta-algorithms*) czy choćby niestandardowe (własne) operatory (ang. *custom operators*).

Przyjrzyjmy się zarysowi niektórych struktur występującym w Marianie:

Back-end

W Marianie zaimplementowano back-end związany z uczeniem głębokim. Opiera się on na automatycznym różnicowaniu w trybie zwrotnym (ang. *reverse-mode auto differentiation*) i grafach obliczeniowych (konstrukcja powyższych zbliżona jest do implementacji w toolkicie DyNet[19]). Back-end systemów podobnych do Mariana poza tłumaczeniem maszynowym mógłby być stosowany również do innych celów. Twórcy Mariana zadecydowali jednak o wprowadzeniu optymalizacji wyłącznie pod kątem realizacji tłumaczenia maszynowego (i innych podobnych mu zadań). W związku z tym wprowadzono na przykład mechanizmy związane z "uwagą" (ang. *attention mechanisms*), czy też zaimplementowano skumulowane neurony RNN (ang. *fused RNN cells*).

Framework Encoder-Decoder

Do wytrenowania modelu tłumacza maszynowego framework Marian stosuje modele sieci neuronowych. Dysponuje np. właściwym dla sieci neuronowych modelem encoder-decoder zaimplementowanym jako dwie klasy – decoder i encoder. W uproszczony sposób prezentuje je kod źródłowy 2.1:

Kod źródłowy 2.1. Klasy decoder i encoder w uproszczonej formie [14]

```
class Encoder {
```

```
    StateOfEncoder build(Batch);
};
class Decoder {
    StateOfDecoder startState(StateOfEncoder[]);
    StateOfDecoder step(StateOfDecoder, Batch);
};
```

Za inspirację do implementacji modelu posłużył framework Moses([2]) służący, podobnie jak Marian, do przeprowadzania zadań związanych z tłumaczeniem maszynowym. Moses opiera się jednak o Statystyczne Tłumaczenie Maszynowe (SMT), podczas gdy Marian uskutecznia Neuronowe Tłumaczenie Maszynowe (NMT). Co więcej pozostawiona została programiście możliwość dodawania kolejnych modeli sieci neuronowych i wykorzystywania ich w dalszym przetwarzaniu.

Meta-algorytmy

We frameworku Marian zaimplementowano efektywne meta-algorytmy. Można wśród nich znaleźć dotyczące takich obszarów jak na przykład:

- rozproszony trening na wielu urządzeniach (zarówno GPU jak i CPU),
- trening na wielu wątkach,
- zastosowanie różnorodnych modeli (np. modele Deep RNN, transformer),
- przeszukiwanie wiązki(ang. *batched beam search*) – np. przetwarzanie ciągów słów.

2.1.2. Struktura danych wejściowych

Marian, narzędzie trenujące modele tłumaczy maszynowych, wymaga źródła informacji w postaci gotowych korpusów (ang. *corpus*) treningowych. Termin "korpus" należy rozumieć jako zbiór zdań (wierszy tekstu), z których każde znajduje się w osobnej linii pliku tekstowego i jako całość stanowią zbiór treningowy modelu tłumacza maszynowego. W przypadku systemu Marian korzysta się z dwóch odpowiadających sobie (mają to samo znaczenie) korpusów językowych: języka źródłowego i języka docelowego. Na potrzeby omawianego tu projektu Magisterskiego wybrano odpowiednio języki niemiecki (fragment na rys. 2.1) i angielski (fragment na rys. 2.2).

Ponadto używane są również zbiory walidacyjne (ang. *validation set*). Podobnie jak w przypadku korpusów stosowane są odpowiadające sobie teksty w języku źródłowym (frag-

```
1 Für die Unstable-Distribution (Sid) wurde dieses Proble
  m in Version 2.0.6-1 behoben.
2 Iran wurde bereits genannt.
3 Teheran ist darum bemüht, die Inhaftierung des Menschen
  rechtsanwalts Abdolfattah Soltani mit dem Argument zu r
  echtfertigen, dass er ein Atomspion sei.
4 Die Band spielte unter anderem auch beim renommierten „
  Warschauer Herbst“.
5 Die konjunkturellen Entwicklungen werden sich dadurch v
  oraussichtlich weiter einander angleichen.
6 Bei der Debatte über das Grünbuch hat sich gezeigt, daß
   bessere Kenntnis der Innovation und der auf diesem Geb
  iet vorhandenen Lücken erforderlich sind.
7 Alexandre Desplat – The Ghost Writer (590 000 EUR)
```

Rys. 2.1. Fragment korpusu (niemiecki)

```
1 For the unstable distribution (sid) this problem has be
  en fixed in version 2.0.6-1.
2 The issue of Iran has already been mentioned.
3 Indeed, Tehran seeks to justify the arrest of human rig
  hts lawyer Abdolfattah Soltani by labelling him as a nu
  clear spy.
4 The band also played venues such as the renowned “Warsa
  w Autumn” festival.
5 Cyclical developments are likely to converge further.
6 The debate centred on the Green Paper has highlighted t
  he need for deeper knowledge of innovation and the pres
  ent shortcomings in this area.
7 Alexandre Desplat - The Ghost Writer (590.000 €)
```

Rys. 2.2. Fragment korpusu (angielski)

ment na rys. 2.3) i w języku docelowym (fragment na rys. 2.4). Zarówno korpusy treningowe jak i zbiory walidacyjne stanowią swego rodzaju wzorzec właściwego tłumaczenia. Wyraźna różnica między nimi (poza treścią, która dla poprawności działania tłumacza musi być różna między zbiorami) to ilość zdań, jakie zawierają — rozmiar zbioru walidacyjnego może być zdecydowanie mniejszy (tutaj po 3 tys. wierszy tekstu dla korpusów mających ok. 1 mln wierszy każdy). Podstawową różnicę stanowi jednak ich przeznaczenie — na podstawie zbiorów treningowych konstruowany jest cały model tłumacza, a zbiory walidacyjne służą do weryfikacji jego poprawności w trakcie treningu.

```
1 Es hat keine Prüfung der Effizienz des Staates, keine E
  inschätzung der wirtschaftlichen Entwicklung, der Situa
  tion an den Grenzen oder der Errichtung echter wirtscha
  ftlicher und politischer Strukturen gegeben.
2 Die Küste Kenias ist ein Paradies mit palmgesäumten Str
  änden und dem warmen einladenden Wasser des Indischen O
  zeans.
3 Art und Herkunftsland der Einrichtungen, die eine EU-Be
  zuschussung beantragen können;
4 Brief des Apostels Paulus an die Korinther (II, 10-10 6
  ) Geschwister: Als seine Mitarbeiter, gibt Wir fordern
  Sie nicht auf die Gnade Gottes vergeblich empfangen Den
  n er sagt: Wenn ich Ihnen und der Tag des Heils hörte i
  ch euch geholfen.
5 Diese Dienste sammeln und verfolgen keine persönlichen
  Informationen.
6 Am 15. Juli fand in unserem Werk in Czerniejewo ein Bet
  riebsfest aus Anlass des zehnjährigen Bestehens von Mul
  timebel statt. Hauptattraktionen waren Wettbewerbe im P
  ingpong-Spielen und Luftgewehrschießen.
7 Das Alona Vida Beach Resort befindet sich im Herzen des
   Alona Beach in einer tropischen Gartenanlage und biete
  t 8 Delux Zimmer mit Aircon, Deckenventilator, warm und
   kalt Wasser (Süsswasser), Kabel- TV mit 34 Kanälen, el
  ektronischem Safe und Terrasse.
```

Rys. 2.3. Fragment zbioru walidacyjnego (niemiecki)

2.1.3. Wyjście

Framework Marian służy przede wszystkim do trenowania modeli tłumaczy automatycznych. W związku z tym rezultatem pracy narzędzia jest plik będący wytrenowanym modelem. Można go z powodzeniem wykorzystać w celu tłumaczenia tekstów w języku źródłowym (w przypadku projektu tekstów niemieckich) na docelowy (tutaj angielski) z wykorzystaniem narzędzia Marian.

2.2. Tree-tagger

Niniejszy rozdział opracowano na podstawie [23].

TreeTagger jest narzędziem służącym do przypisywania tagów oznaczających części mowy do wyrazów w tekście (wcześniej wykonuje tokenizację) oraz znajdowania lematów dla poszczególnych słów. Poniżej zostaną pokazane jego podstawowe funkcje oraz zostanie przybliżony sposób użycia. Następnie zostanie przedstawiony program TreeTagger Python

```
1 There has been no review of the efficiency of the state
  , no assessment of economic development, of the situati
  on at the borders or of the building of genuine economi
  c and political structures.
2 The Kenyan Coastline is a paradise of palm fringed beac
  hes and the warm inviting waters of the Indian Ocean .
3 the type and country of origin of organisations that ma
  y apply for EU funding;
4 Letter from St. Paul the Apostle to the Corinthians (II
  , 10-10 6) Siblings: As his employees, there We urge yo
  u not to receive the grace of God in vain For he says:
  When I have heard you and the day of salvation I have h
  elped you.
5 These services do not collect or track any personal inf
  ormation.
6 On the 15th July, on the occasion of the 10th anniversa
  ry of Multimebel, an integration meeting in the grounds
   of the plant in Czerniejew took place.
7 Alona Vida Beach Resort is located in the middle of Alo
  na Beach surrounded by a tropical garden and offers 8 d
  eluxe rooms with aircon, fan, warm and cold water (fres
  hwater), cable TV with 34 channels, electronic safe and
   terrace.
```

Rys. 2.4. Fragment zbioru walidacyjnego (angielski)

Wrapper umożliwiający korzystanie z TreeTaggera za pośrednictwem języka Python.

2.2.1. Podstawowe informacje

Do zastosowań TreeTaggera należą opisywanie poszczególnych słów tekstu za pomocą skompresowanych informacjach dotyczących części mowy oraz lematów. Narzędzie opracował Helmut Shmid w ramach projektu *TC project* w Instytucie Lingwistyki Komputerowej na Uniwersytecie w Stuttgarcie (Computational Linguistics of the University of Stuttgart). Do języków, dla których można z powodzeniem wykorzystać Tnarzędzie TreeTagger należą między innymi: angielski, francuski, niemiecki, rosyjski, swahili, polski i wiele innych. W przypadku konieczności obsługi języka spoza zaimplementowanych domyślnie, programiście pozostaje możliwość rozbudowania funkcjonalności narzędzia o nowe języki. Należy wtedy dostarczyć nowy korpus treningowy oraz leksykon.

2.2.2. Struktura danych wejściowych

W związku z faktem, że TreeTagger wykonuje operacje dotyczące konkretnego tekstu, właśnie konkretny zbiór tekstowy stanowi jego dane wejściowe. Nie jest przy tym istotnie, czy

tekst został uprzednio ztokenizowany — dla języków takich jak polski oraz niemiecki czy angielski (dwa ostatnie używane w niniejszym projekcie) tokenizacji dokonuje TreeTagger.

2.2.3. Wyjście

```
marek@cerebrum:~$ echo "Is he doing  it right?" \
> | tree-tagger-english
        reading parameters ...
        tagging ...
         finished.
Is      VBZ     be
he      PP      he
doing   VBG     do
it      PP      it
right   RB      right
?       SENT    ?
```

Rys. 2.5. Tekst z tagami - przykład

Przykład działania TreeTaggera na angielskim zdaniu *Is he doing it right?* widoczny jest na rysunku 2.5. Od napisu *reading parameters*, w trzecim wierszu wyniku, widoczne są komunikaty wypisywane w konsoli przez narzędzie. Sam rezultat działania zaczyna się w wierszu poprzedzanym przez informację $finished.$. Pierwsza kolumna zawiera oryginalne słowo zdania podanego na wejście (np *right*). W drugiej kolumnie zawarty jest znacznik przyporządkowany danemu słowu (w wypadku słowa *right* jest to tag *RB* czyli *adverb* – przysłówek). Trzecia kolumna zawiera lemat właściwy dla danego słowa (słowo *right* przypisane jest do lematu *right*).

2.2.4. TreeTagger Python Wrapper

Niniejszy rozdział opracowano na podstawie Pointal L. [21].

TreeTagger Python Wrapper jest narzędziem umożliwiającym korzystanie z funkcjonalności narzędzia TreeTagger za pośrednictwem języka programowania Python. W projekcie używano go do inicjalizacji tagowania (z tokenizacją) dla zbiorów języka angielskiego oraz do tokenizacji tekstów języka niemieckiego.

Metody

Poniżej opisane zostaną najważniejsze metody używane na potrzeby projektu.

TaggerProcessPoll — klasa (i jej konstruktor) z biblioteki *treetaggerpoll*

Aby umożliwić wykonanie przetwarzania na różnych wątkach, metoda (konstruktor klasy) tworzy sondaż (ang. *poll*) procesu Treetaggera. Wszystkie procesy zebrane w jednym sondażu wykonują równolegle to samo zadanie (tokenizacja/chunking i tagowanie), na przykład w przypadku każdego procesu wszystkie opcje tagowania są takie same.

Jeśli chodzi o przetwarzanie obiektów klasy $TaggerProcessPoll$, nazwy metod wchodzących w skład jej interfejsu kończą się sufiksem *_async*. Poza tym całe ich nagłówki są takie same jak w analogicznych metodach klasy $TreeTagger$. Każda z nich zwraca obiekt *ProcJob*, który umożliwia rozpoznanie zakończenia przetwarzania.

Kod źródłowy 2.2. TaggerProcessPoll — nagłówek

```
1   TaggerProcessPoll(workerscount=None, keepjobs=True,
        wantresult=True, keeptagargs=True, **kwargs)
```

Aby utworzyć obiekt $TreeTaggerPoll$ należy wywołać konstruktor klasy (Kod źródłowy 2.2) i określić jej parametry (tabela 2.1).

Tab. 2.1. wszystkie parametry metody $TaggerProcessPoll$

parametr	typ wartości
workerscount	int
keepjobs	boolean
wantresult	boolean
keeptagargs	boolean
kwargs	boolean

Znaczenie poszczególnych parametrów opisano poniżej:

workerscount — domyślnie $None$;

oznacza liczbę procesów realizujących zadanie w ramach jednego sondażu (jeśli parametr przyjmie wartość $None$, to zostanie utworzona liczba procesów zgodna z maksymalną liczbą wątków dostępnych dla procesora komputera, na którym zachodzi całość przetwarzania),

keepjobs — domyślnie $True$;

określa, czy sondaż ma przechowywać referencje do poszczególnych zadań, aby zarządzać wynikami od realizujących je (to zadanie) procesów i zwracać wyniki ich przetwarzania,

wantresult — domyślnie *True*;

określa, czy poszczególne procesy mają zwracać wyniki przetwarzania w celu zebrania ich w ramach zadania,

keeptagargs — domyślnie *True*;

określa, czy w obiekcie synchronizacji *ProcJob* mają być przechowywane argumenty tagowania,

kwargs — kilka parametrów dotyczących przetwarzania realizowanego w ranach sondażu (w projekcie manipulowano tylko zaliczającym się do nich parametrem *TAGLANG* określającym język jakiego ma dotyczyć przetwarzanie).

Po zakończeniu przetwarzania, aby poprawnie zamknąć otwarty wcześniej poll, wywołuje się metodę *TaggerProcessPoll.stop_poll*() — rozwiązuje ona *TaggerProcessPoll*.

tag_text_async

Metoda *tag_text_async* wykonuje swoje działanie na obiektach klasy *TreeTaggerPoll*. Służy do współbieżnego tagowania tekstu. jej działanie można uzależnić od wartości parametrów, które zostały zebrane w tabeli 2.2 wraz z typem wartości, jaki każdy z nich przyjmuje.

Tab. 2.2. Wszystkie parametry metody *tag_text_async*

parametr	**typ wartości**
text	unicode string / [unicode string]
numlines	boolean
tagonly	boolean
prepronly	boolean
tagblanks	boolean
notagurl	boolean
notagemail	boolean
notagip	boolean
notagdns	boolean
nosgmlsplit	boolean

Znaczenie wyszczególnionych w tabeli parametrów opisano poniżej:

text — tekst, jaki ma zostać otagowany,

numlines — domyślnie *False*;

wskazuje, czy zachować numerowanie linii w trakcie przetwarzania,

tagonly — domyślnie *False*;

zaznacza, czy ma zostać wykonane wyłącznie tagowanie (w przypadku wartości *True* wymagane jest przekazanie do modułu tekstu zawierającego po jednym tokenie w danym wierszu tekstu),

prepronly — domyślnie *False*;

ustala, czy ma zostać przeprowadzony wyłącznie preprocessing (bez tagowania),

tagblanks — domyślnie *False*;

określa, czy ma zostać zachowana informacja o pustych znakach, stosuje się wtedy tagi SGML,

notagurl — domyślnie *False*;

wskazuje, czy adresy URL nie mają zostać zastąpione przez domyślny dla nich znacznik,

notagemail — domyślnie *False*;

specyfikuje, czy adresy email nie mają zostać zastąpione przez domyślny dla nich znacznik,

notagip — domyślnie *False*;

orzeka, czy adresy IP mają zostać zastąpione przez domyślny dla nich znacznik,

notagdns — domyślnie *False*;

zaznacza, czy nazwy DNS mają zostać zastąpione przez domyślny dla nich znacznik,

nosgmlsplit — domyślnie *False*;

określa, czy mają zostać rozdzielone tagi SGML obecne w tekście źródłowym.

make_tags

Funkcja należąca do *treetaggerwrapper*. Przekształca otagowane dane tekstowe powstałe po przetworzeniu tekstu funkcją *TreeTagger.tag_text* (albo opisaną wyżej *tag_text_async*) na listę nazwanych krotek (ang. *tuple*). Każdy jej element można określić jako *Tag*, *TagExtra* albo *NotTag*. Krotki typu *Tag* i *TagExtra* dysponują atrybutami *word*, *pos* i *lemma*. Funkcja przyjmuje trzy parametry:

result — wynik przetworzenia tekstu funkcją *TreeTagger.tag_text* (albo analogiczną *tag_text_async*),

exclude_nottags — *bool* (domyślnie wartość *False*) ;

nie generuj krotki *NotTag* dla wyników o złym rozmiarze,

allow_extra — *bool* (domyślnie wartość $False$);

tworzy $TagExtra$ dla wyników dłuższych niż oczekiwano.

Lista otrzymywana w wyniku przetwarzania funkcji ułatwia dalsze przetwarzanie otagowanego tekstu za pomocą narzędzi języka Python.

Rozdział 3

Przetwarzanie wstępne

Trening tłumacza automatycznego wymaga właściwie przygotowanych zbiorów tekstowych. Poniższe podrozdziały przedstawiają realizujące to zadanie procesy gromadzenia i ekstrakcji danych treningowych (rozdział 3.1), tokenizacji (rozdział 3.2), truecasingu (rozdział 3.3.1) oraz aplikacji kodowania BPE (rozdział 3.3.2).

3.1. Początkowe operacje

Przed rozpoczęciem treningu modelu tłumacza maszynowego na korpusach trenujących, muszą one zostać odpowiednio przygotowane. Z dostępnego zbioru zdań wybierana jest jego reprezentatywna część. Dopiero po procesach przycinania (ang. *shortening*), czyszczenia (ang.*cleaning*), tokenizacji i tagowania wykorzystuje się ją do treningu modelu tłumacza maszynowego. Cały proces przygotowania danych przebiega według przedstawionego poniżej algorytmu (Algorytm 1).

3.1.1. Gromadzenie danych tekstowych

Korpus używany do treningu modelu tłumacza powinien być dostosowany do przyszłych zastosowań translatora, do rodzaju tekstów jakie będzie tłumaczył (np. treści w języku urzędowym). Sytuacja, w której nie jest wiadome jaki będzie charakter przetwarzanego tekstu, skłania do dostosowania tłumacza zarówno do wypowiedzi potocznych, jak i do tekstów formalnych, urzędowych. Z tego powodu w korpusie treningowym wskazane jest użycie tak samo dużego zbioru zdań dla każdego z powyższych stylów wypowiedzi. W omawianym tu projekcie magisterskim skorzystano z korpusów

Algorytm 1: Preprocessing

Input: Surowe dane tekstowe
Output: Model tłumacza automatycznego
Data: Zbiory tekstowe a1, a2, a3, a4

```
raw = polacz dane z roznych zrodel (a1, a2, a3, a4)
shuf = shuffle(raw)
short = zmniejsz ilosc danych(shuf)
clean = przytnij(short)
norm = normalizuj punctuation(clean)
tok = tokenizuj(norm)
en, tag = taguj korpus angielski(tok)
true = truecase(en)
merged = polacz(true,tag)
```

commoncrawl, europarl-v7, news-commentary-c12 i rapid2016 z zasobów konkursowych wmt2017 (`http://www.statmt.org/wmt17/translation-task.html`) konferencji Empirical Methods in Natural Language Processing udostępnionych w systemie Gonito (`https://gonito.net/challenge/wmt-2017`).

Kod źródłowy 3.1. Gromadzenie i łączenie korpusu

```
### Aggregate corpse parts
echo "Aggregating corpse parts..."
cat $train/commoncrawl.tsv \
$train/europarl-v7.tsv \
$train/news-commentary-v12.tsv \
$train/rapid2016.tsv \
> $corpseDir/rawCorpse.tsv
echo "done\n"

### Shuffle lines
echo "Shuffling lines"
shuf $corpseDir/rawCorpse.tsv > $corpseDir/rawCorpse.shuf.tsv
echo "done\n"
```

W kodzie źródłowym 3.1 zamieszczone zostały polecenia użyte do połączenia czterech plików tekstowych w jeden stanowiący bazę do kolejnych operacji. Na początku *cat* tworzy strumień składający się z zawartości czterech plików, których nazwy (ścieżki dostępu) przekazano w parametrach polecenia. Wynik działania polecenia (strumień tekstowy) nie jest jednak wyświetlany w oknie terminala (standardowe wyjście dla polecenia *cat*) ale zostaje zapisany w pliku.

3.1.2. Skracanie korpusów

Framework Marian tworzy modele tłumaczy automatycznych na podstawie korpusów języka źródłowego i docelowego. Dla miarodajnych wyników potrzebne są jak największe zbiory tekstowe. Niestety ich rozmiar jest ograniczony z góry przez możliwości sprzętu, na którym zachodzi przetwarzanie. Z dużego zbioru tekstu można więc wyodrębnić korpusy o zadanej wielkości.

Kod źródłowy 3.2. Ekstrakcja korpusów treningowych

```
1 ### Shorten corpses:
2 echo "Shortening corpses..."
3 head -n $corpseLen $corpseDir/rawCorpse.shuf.de \
4     > $corpseDir/corpse$folderName.$src
5 head -n $corpseLen $corpseDir/rawCorpse.shuf.en \
6     > $corpseDir/corpse$folderName.$tgt
7 echo "done\n"
```

Kod źródłowy 3.2 przedstawia operacje, w wyniku których korpusy języka źródłowego (niemiecki) i docelowego (angielski) zostają odpowiednio "przycięte". Polecenie *head* pobiera pierwsze $corpseLen linii przetwarzanego pliku do strumienia. Jego zawartość zostaje następnie przekierowana do pliku $corpse\$folderName.\$src$ (albo $corpse\$folderName.\$tgt$). Używane w tym zapisie zmienne *corpseLen*, $\$folderName$ i $\$src$ oznaczają odpowiednio:

- *corpseLen* - rozmiar korpusu (liczba zdań/wierszy jakie zawiera),
- $\$folderName$ - nazwa folderu przechowującego nowy korpus,
- $\$src$/$\$tgt$ - język źródłowy/docelowy, odpowiednio niemiecki (de) i angielski (en).

Kod źródłowy 3.3. Ekstrakcja zbiorów walidacyjnych

```
1 # Extracting validation set
2 echo "Extracting validation set..."
3 tail -n $validSetLen $corpseDir/rawCorpse.shuf.de \
4     > $corpseDir/validSet.$src
5 tail -n $validSetLen $corpseDir/rawCorpse.shuf.en \
6     > $corpseDir/validSet.$tgt
7 echo "done\n"
```

Podobnie sytuacja wygląda w przypadku pozyskiwania zbioru walidacyjnego. Kod źródłowy 3.3 zawiera skrypt realizujący tę operację. Polecenie *tail* pobiera ostatnie $*validSetLen* linii przetwarzanego pliku do strumienia. Jego zawartość zostaje następnie przekierowana do pliku *validSet*.$*src* (albo *validSet*.$*tgt*). Zmienne $*src* i $*tgt* oznaczają jak wyżej *de* i *en* (języki niemiecki i angielski), natomiast zmienna $*validSetLen* określa rozmiar zbioru walidacyjnego (liczbę zdań w tym zbiorze).

3.1.3. Czyszczenie korpusów (ang. *cleaning*)

Aby zgromadzony zbiór tekstowy mógł działać poprawnie z wykorzystywanymi skryptami trenującymi, potrzebne jest jego czyszczenie (*cleaning*). W skład czynności, które ono oznacza, wchodzi:

- usuwanie pustych lini (linie nie zawierające tekstu),
- usuwanie linii zbyt krótkich lub zbyt długich,
- usuwanie nadmiarowych spacji (odstępów) w tekście.

W projekcie wykorzystano skrypt $clean-corpus-n.perl$ pochodzący z pakietu Moses ([2]). Wzorzec jego wywołania wygląda następująco:

clean-corpus-n.perl CORPUS L1 L2 OUT MIN MAX.

Przyjmowane parametry przedstawione są w tabeli 3.1

Tab. 3.1. parametry programu clean-corpus-n.perl

parametr	**opis**
CORPUS	pliki nazwa plików tekstowych których ma dotyczyć proces czyszczenia (bez rozszerzeń właściwych dla danych języków)
L1	język źródłowy (skrót w rodzaju de)
L2	język docelowy (skrót w rodzaju en)
OUT	nazwa plików które powstaną w wyniku działania skryptu (zostaną do nich dopisane użyte rozszerzenia .L1 i .L2)
MIN	minimalna długość zdania (krótsze są wycinane)
MAX	maksymalna długość zdania (dłuższe są wycinane)

W przypadku realizowanego parametru zmienne przyjmują następujące wartości:

- CORPUS - $corpseDir/corpus.np,
- L1 - de (wartość $src),
- L2 - en (wartość $tgt),
- OUT - $corpseDir/corpus.clean,
- MIN - 1,
- MAX - 1024 ($maxSenLen).

Kod źródłowy 3.4. czyszczenie korpusów

```
1 ### Cleaning
2 echo "Cleaning..."
3 $mosesScr/training/clean-corpus-n.perl $corpseDir/corpus.np $src
     $tgt $corpseDir/corpus.clean 1 $maxSenLen
     $corpseDir/corpus.retained
4 echo "done\n"
```

Wywołanie skryptu wykonującego w realizowanym projekcie operacje *czyszczenia* potrzebnych korpusów prezentuje Kod źródłowy 3.4. Zmienna $*maxSenLen* oznacza tutaj maksymalną dopuszczalną długość zdania (MAX w tabeli 3.1)

3.2. Tokenizacja

Niniejszy rozdział opracowano na podstawie Koehn P.[15]

Ważny element wstępnego przygotowania korpusu do dalszego przetwarzania stanowi tokenizacja. Jest to podział surowego tekstu na uporządkowany zbiór mniejszych części, nazywanych tokenami. W zależności od potrzeb tokenem może być akapit w zbiorze całego tekstu, zdanie w akapicie, słowo w zdaniu a nawet litera w słowie.

TEXT: Furthermore, this student's project could be highly-usable.
TOKENS: "Furthermore" "," "this" "student" "'s" "project" "could" "be" "highly-usable" "."

Rys. 3.1. Tokenizacja

Dla omawianego przypadku tłumaczenia maszynowego tokenizacja to proces podziału jednolitego tekstu na listę słów (Rys. 3.1). Tokenem jest wtedy pojedyncze słowo lub znak interpunkcyjny.

Niestety w przypadku realizacji tokenizacji występuje kilka problemów. Pierwszy pojawia się dla języków nie uwzględniających przerw (spacji) między kolejnymi słowami. Taka trudność występuje na przykład dla języka chińskiego. W celu oddzielania kolejnych słów (wyodrębnienia tokenów) potrzebny jest w takim przypadku zestaw metod umożliwiających rozstrzygnięcie możliwych dwuznaczności i zarazem leksykon wszystkich możliwych słów.

TEXT: Furthermore, this student's project could be highly-usable.
TOKENS: "Furthermore," "this" "student's" "project" "could" "be" "highly-usable."

Rys. 3.2. Naiwna tokenizacja

Dla języków zapisywanych w alfabecie łacińskim, np przykład języka angielskiego (rys. 3.2.), podział tekstu na podstawie spacji wydaje się relatywnie prosty do zrealizowania. W rzeczywistości również w przypadku takich języków pojawiają się pytania nie mające jednoznacznych odpowiedzi.

- Nie jest oczywiste co zrobić z konstrukcjami w rodzaju *student's* czy *didn't*. Zarówno dzierżawczy przyrostek (ang. *possessive marker*) jak i połączone słowa można potraktować jak jeden token (rys. 3.2.). Z drugiej strony można by też oddzielić słowo od części występującej po znaku " ' " i utworzyć dwa osobne tokeny.

- Problem stanowi kwestia tokenizacji wyrazów zbudowanych z dwóch (i więcej) słów połączonych myślnikiem. Określenia w rodzaju *co-funder* można ztokenizować jako jedno słowo. Z drugiej jednak strony, w przypadku określeń jak *low-risk*, utworzenie kilku niezależnych tokenów wydaje się również uzasadnione.

Tokenizację przeprowadza się, aby móc traktować tekst jako zbiór tokenów. Co więcej, przetwarzanie tekstów (w szczególności trening modeli tłumaczy maszynowych), potrzebuje możliwie małego zbioru różnych tokenów, który pozwala na reprezentowanie jak największego zbioru tekstowego. Nie ma potrzeby przechowywania wszystkim możliwych połączeń w rodzaju *student's* czy *'house'*. Nie trzeba też uczyć modelu tłumacza wielu różnych wersji translacji słów w zależności od tego, czy np. występują one z przecinkiem albo czy zostały zawarte w cudzysłowie (Rys. 3.2).

Problem z rozdzielaniem wyrazów zachodzi nie tylko w przypadku wyrazów z myślnikiem, dotyczy również wyrazów powstałych z kilku bezpośrednio połączonych słów. Dotyczy on głównie języków takich jak język niemiecki.

raw text: Kindergarten
raw text: weltschmerz
raw text: Maschinenübersetzung

Rys. 3.3. Słowa niemieckie

Nowe wyrazy często powstają w nim przez sklejenie dwóch (lub więcej) niezależnie istniejących słów. Przykładami mogą być *kindergarten* (przedszkole), *weltshmerz* (znużenie światem) czy *Maschinenübersetzung* (tłumaczenie maszynowe). W związku z tym, że każda ze składowych wyrazu może z powodzeniem pełnić samodzielną rolę (jest samodzielnym słowem) niektóre tokenizatory tworzą w ich przypadku osobne tokeny (Rys. 3.3).

3.2.1. Implementacja w projekcie

Tokenizacja występuje jako integralna część niektórych pakietów przetwarzających tekst. Sytuacja taka zachodzi również w przypadku używanego tutaj narzędzia *TreeTagger* (podrozdział 2.2) stosowanego za pośrednictwem biblioteki *treetaggerwrapper* (podrozdział 2.2.4). W realizowanym projekcie dla tekstu w języku niemieckim posłużył on właśnie do tokenizacji. Pełny skrypt realizowanych operacji zamieszczono w dodatku A (kod źródłowy A.1). Poniżej omówiono jego wybrane fragmenty.

Kod źródłowy 3.5. Niemieckie zbiory tekstowe

```
11 corpse_dir1 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.clean.de'
12 corpse_dir2 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.np.de'
13 corpse_dir3 = '/home/marek/master/gonito/wmt-2017/dev-0/in.np.tsv'
14 corpse_dir4 = '/home/marek/master/gonito/wmt-2017/dev-1/in.np.tsv'
15 corpse_dir5 = '/home/marek/master/gonito/wmt-2017/test-A/in.np.tsv'
16 corpse_dir = [corpse_dir1, corpse_dir2, corpse_dir3, corpse_dir4,
       corpse_dir5]
```

W kodzie źródłowym 3.5 zebrane zostały niemieckie zbiory tekstowe używane w dalszym przetwarzaniu. Wśród nich są (wiersze 11-15 kodu 3.5): korpus (w języku niemiec-

kim), zbiór walidacyjny i zbiory testowe (na tych ostatnich przeprowadzano następnie ewaluację tłumaczeń otrzymanych przy użyciu modeli).

Kod źródłowy 3.6. Tokenizacja – metoda *start_test*

```
27 def start_test(n=None):
```

Zasadnicza część działań zachodzi w metodzie *start_test* (kod 3.6).

Kod źródłowy 3.7. Tokenizacja – importowanie bibliotek

```
29     import treetaggerpoll as ttp
30     import treetaggerwrapper as ttw
```

Importowane są potrzebne biblioteki (kod 3.7) i dla każdego używanego zbioru faktyczne przetwarzanie zaczyna się of Lini 38 (kod 3.8).

Kod źródłowy 3.8. Tokenizacja – metoda *TaggerProcessPoll*

```
38         p = ttp.TaggerProcessPoll(workerscount=n, TAGLANG="de")
```

Zawiera ona metodę tworzącą sondaż (ang. *poll*) dla określonej liczby ($workerscount = n$) procesów wykonujących w jego obrębie zadanie (ang. *workers*) dla wybranego języka. Wartość n ustala się w miejscu wywołania metody, domyślnie wartość ta jest zgodna z maksymalną liczbą wątków dostępnych dla danego procesora. Jeśli chodzi o obsługę konkretnego języka, to używa się właściwego jemu skrótu (tutaj wybrano język niemiecki: *de*).

Dalsze przetwarzanie można podzielić na dwa etapy:

1. tokenizacja wszystkich zdań (linii) przetwarzanego pliku tekstowego (kod źródłowy 3.9 i 3.10, wiersze od 41 do 49),
2. zapisanie otrzymanych sekwencji tokenów do pliku wyjściowego (kod źródłowy 3.11, wiersze od 53 do 72).

Kod źródłowy 3.9. Tokenizacja – operacja *open*

```
41         with open(cd, 'r') as in_file:
```

Podstawowym etapem przetwarzania jest oczywiście część odpowiedzialna za tokenizację. Na początku używany plik zostaje otwarty "do odczytu" i od razu zostaje przypisany do zmiennej in_file (kod 3.9). Następnie dla każdego zdania w otwartym pliku (pętla for kod 3.10) zostaje przeprowadzona tokenizacja.

Kod źródłowy 3.10. Tokenizacja – metoda *tag_text_async*

```
46              for sentence in in_file.readlines():
47                  # print('sen:'+sentence, end='')
48                  taggSet.append(p.tag_text_async(sentence,
                        prepronly=True, notagurl=True,
49                                                  notagemail=True,
                                                        notagip=True,
                                                        notagdns=True,
                                                        nosgmlsplit=True))
```

Do tego celu zostaje wykorzystana metoda *tag_text_async* opisana w rozdziale 2.2.4. Wymagane jest tutaj ograniczenie jej działania wyłącznie do tokenizacji oraz zapewnienie poprawnego podziału wyrazów i znaków interpunkcyjnych na tokeny. Zastosowano więc odmienne od domyślnych wartości parametrów metody — tabela 3.2 (por. tabela 2.2).

Tab. 3.2. Parametry metody *tag_text_async* dla tokenizacji zbiorów

parametr	wartość
text	sentence
prepronly	True
notagurl	True
notagemail	True
notagip	True
notagdns	True
nosgmlsplit	True

Wpływ wyszczególnionych parametrów (a także innych sterujących podaną metodą) opisano w rozdziale 2.2.4 wraz z metodą. W tym miejscu na szczególną uwagę zasługuje parametr *prepronly*. Domyślnie przyjmuje on wartość $False$. Ze względu jednak na to, że przetwarzane są teksty w języku niemieckim, nie ma potrzeby opisywania słów za pomocą tagów. Stąd zastosowano $prepronly = True$ (linia 48 kodu 3.10).

Tokenizacja wszystkich wykorzystywanych zdań powyższą metodą skutkuje otrzymaniem listy list tokenów(po jednej liście tokenów dla każdego zdania/ linii). Nie wystarcza to jednak do potrzeb dalszego przetwarzania. Niezbędny jest plik tekstowy zawierający stokenizowane zdania (po jednym zdaniu w każdym wierszu). Aby utworzyć potrzebny plik realizowany jest drugi z wyszczególnionych wcześniej etapów.

Kod źródłowy 3.11. Tokenizacja – zapis zdań do pliku

```
53          with open(tagged_dir[j], 'w') as out_file:
54              print(tagged_dir[j]+' opened to write')
```

```
55              print('Make tokens & write file...')
56              for i, r in enumerate(taggSet):
57                  r.wait_finished()
58                  # print(i, str(r.result))
59                  currLine = currLine+1
60
61                  for token in r.result:
62                      # print(i, token+' ', end='')
63                      out_file.write(token+' ')
64                  # print()
65                  out_file.write('\n')
66
67                  if (currLine == PACK_SIZE):
68                      currLine = 0
69                      print('line tokenized:', i+1, '\tfirst word:',
                            r.result[0])
70
71                  taggSet[i] = None
72              print('Tokenized', i+1, 'lines')
```

Kod będący implementacją wykonanych operacji zamieszczono w kodzie źródłowym 3.11. Na początku potrzebne jest otwarcie pliku "do zapisu", w nim będą przechowywane stokenizowane zdania. Realizuje to funkcja *open* (linia 53. w kodzie 3.11), a zwracana przez nią wartość jest przypisana do zmiennej out_file. Następnie w linii 56. rozpoczyna się zapisywanie poszczególnych zdań (lini) w stokenizowanej formie. Na początku przetwarzanie jest wstrzymane do momentu, w którym tokenizacja zdania zostanie zakończona (tag_text_async działa asynchronicznie w różnych procesach), wykonuje to funkcja $r.wait_finished$ w lini 57.

Kod źródłowy 3.12. Tokenizacja – operacja *write*

```
61                  for token in r.result:
62                      # print(i, token+' ', end='')
63                      out_file.write(token+' ')
64                  # print()
65                  out_file.write('\n')
```

Dalej rozpoczyna się pętla for, wewnątrz której dokonany zostaje zapis wydzielonych wcześniej tokenów do pliku. Metoda $out_file.write(...)$ zapisuje zawartość nawiasu (...) do pliku określonego zmienną out_file. Forma wpisywanego tekstu określona jest jako

$token + ' '$, co oznacza kolejny przetwarzany token w wierszu i następującą po nim spację. W ten sposób w pliku docelowym każdy wiersz składa się z sekwencji tokenów oddzielonych spacją. Instrukcja $out_file.write('\backslash n')$ występująca zaraz po pętli powoduje przejście do nowej lini (dodanie znaku nowej lini) po ukończeniu wpisywania tokenów z przetwarzanego zdania.

Kod źródłowy 3.13. Tokenizacja – wyświetlanie komunikatów dot. przetwarzania

```
67                  if (currLine == PACK_SIZE):
68                      currLine = 0
69                      print('line tokenized:', i+1, '\tfirst word:',
                            r.result[0])
```

.

Komentarza może wymagać pętla for w liniach 67-69 (kod źródłowy 3.13). Jest ona zaledwie formą informowania użytkownika o numerze zapisywanego w danym momencie wiersza. Stała $PACK_SIZE$ odpowiada za krok z jakim wypisywane są w konsoli kolejne informacje o stanie przetwarzania (krok ten wyrażony jest w liczbie przetworzonych wierszy).

Kod źródłowy 3.14. Tagowanie i tokenizacja - uruchomienie skryptu

```
1  ### Tagging with Tokenization
2  echo "TAGGING and tokenization"
3  for lang in $src $tgt; do
4      echo "$lang"
5      chmod 777 $workingDir/tag-$folderName-$lang.py
6      python3 $workingDir/tag-$folderName-$lang.py    #
           corpus.tg.$lang
7  done
8  echo "done tagging and tokenization\n"
```

Kod źródłowy 3.14 przedstawia uruchomienie skryptów realizujących tokenizację i tagowanie używanych zbiorów tekstowych (w przypadku języka nienieckiego jest to wyłącznie tokenizacja). Zmienne *src* i *tgt* oznaczają tutaj odpowiednio język źródłowy — angielski (*en*) i język docelowy — niemiecki (*de*). Wykonywane są dwa przebiegi pętli for, dla których zmienia się wartość zmiennej *lang*. Przypisane do niej zostają zostają kolejno wartości zmiennych *src* i *tgt*. Oznaczają one odpowiednio język źródłowy (wartość *de*) i język docelowy (wartość *en*). Dalej wobec wykonywanego później skryptu języka Python ($\$folderName - \$lang.py$, gdzie wartość *lang* jest uzależniona od numeru przebiegu) wy-

konane zostaje polecenie systemu Linux *chmod*, stosowany do zmiany zezwoleń dostępu do plików — wartość 777 oznacza, że grupa, właściciel i pozostali użytkownicy pliku mają możliwość jego odczytu, zapisu (modyfikacji) i wykonania. Od linii 6. kodu 3.14 przetwarzany jest skrypt języka Python wskazany jako argument polecenia *python*3.

3.3. Normalizacja słów oraz algorytm BPE

Przygotowanie zbiorów tekstowych do przetwarzania przed treningiem translatora wiąże się z normalizacją słów zawartych w tekstach oraz z przynajmniej minimalną kompresją danych. Zajmujące się tym procesy opisano w sekcjach 3.3.1 i 3.3.2.

3.3.1. Truecasing

Niniejszy rozdział opracowano na podstawie Koehn P. [15].

Mając na uwadze minimalizację przechowywanego zbioru tokenów rozróżnianie słów w zależności od wielkości tworzących je liter jest zazwyczaj niewskazane. Nie ma potrzeby szukania osobnych tłumaczeń na przykład dla wyrazów *Machine*, *machine* czy *MACHINE*. Każde ich wystąpienie odpowiednio na początku, w środku zdania czy w tytule, nie powinno wpływać na tłumaczenie słowa. Aby to osiągnąć, podobnie jak w przypadku tokenizacji, dąży się do minimalizacji przechowywanego słownika. Na przetwarzanych tekstach wykonuje się więc zwykle jedną z poniższych operacji:

- lowercasing,
- truecasing.

Lowercasing jest przetworzeniem tekstu które zamienia wszystkie jego litery na małe zachowując przy tym oryginalną treść tekstu (słowa pozostają takie same).

Truecasing podobnie jak lowercasing zachowuje oryginalną treść tekstu.

Truecasing - podobnie jak lowercasingu normalizuje wielkość liter w słowie. Nie zamienia jednak absolutnie wszystkich liter na małe. Zachowuje wielkie litery na początku imion i nazwisk.

Implementacja

Przy wykorzystaniu odpowiednich narzędzi przeprowadzenie *truecasingu* sprowadza się do wykonaniu kilku operacji. Należy we właściwy sposób (z adekwatnymi argumentami wywołać odpowiednie programy lub wykorzystać skrypty. Wszystkie potrzebne polecenia zawiera kod źródłowy 3.15.

Kod źródłowy 3.15. Trenowanie i przeprowadzenie normalizacji słów

```
31 echo "Train truecaser & truecase corpse"
32 for lang in $src $tgt; do
33     echo "training truecaser .$lang..."
34     $mosesScr/recaser/train-truecaser.perl -model
           $corpseDir/truecase-model.$lang -corpus
           $corpseDir/corpus.text.$lang
35     echo "done"
36     echo "truecasing corpus.text.$lang..."
37     $mosesScr/recaser/truecase.perl < $corpseDir/corpus.text.$lang
           > $corpseDir/corpus.text.tc.$lang -model
           $corpseDir/truecase-model.$lang
38     echo "done"
39     echo "truecasing validSet.text.$lang..."
40     $mosesScr/recaser/truecase.perl <
           $corpseDir/validSet.text.$lang >
           $corpseDir/validSet.text.tc.$lang -model
           $corpseDir/truecase-model.$lang
41     echo "done"
```

Przed przeprowadzeniem normalizacji słów (ang. *truecasing*) należy wytrenować model dla danego języka. Służy do tego skrypt *train−truecaser.perl* dostarczony wraz z pakietem Marian a utworzony pierwotnie dla pakietu Moses. Jego wywołanie znajduje się w lini 34. zamieszczonego powyżej kodu. Parametr *−model* oznacza nazwę pliku tekstowego do jakiego zapisany zostanie model dotyczący przetwarzanego tekstu. Sam wejściowy plik tekstowy przekazywany jest jako wartość parametru *−corpus*.

Dysponując wytrenowanym modelem można przeprowadzić *truecasing* na używanych zbiorach tekstowych. Kod źródłowy 3.15 w wierszach 37 i 40 prezentuje zastosowanie modelu do ujednolicenia wielkości liter w korpusach poszczególnych języków (źródłowy *de*, docelowy *en*) a także na odpowiadających im zbiorach walidacyjnych. Wykorzystuje się w tym wypadku skrypt *truecase.perl* analogicznie do kodu trenującego, pochodzący

z pakietu *Moses*. Podobnie, jak to było w przypadku treningu, jako wartość parametru *model* zostaje podana ścieżka dostępu do wytrenowanego wcześniej modelu *truecasing*u. Sam plik tekstowy, na którym ma zostać przeprowadzony *truecasing*, zostaje przekazany do skryptu za pośrednictwem potoku terminala systemu Linux.

3.3.2. Byte Pair Encoding

Niniejszy rozdział opracowano na podstawie Sennrich et.al.[26]

Dążenie do maksymalnego ograniczenia rozmiaru danych wejściowych dla treningu modelu tłumacza maszynowego przyczynia się do wprowadzenia podstawowej kompresji danych. Jako przyjętą metodę stosuje się tutaj *Byte Pair Encoding* (Gage[11]) nazywaną dalej w skrócie BPE. W oryginalnej wersji BPE jest prostą techniką kompresji danych, którą zaadaptowano do potrzeb segmentowania słów w tekście (rozdział 3.2 pracy [26]). Wykorzystuje się tutaj podział przetwarzanego tekstu na jednostki o rozmiarze litery (pojedynczego znaku). Podstawowy schemat wyodrębniania segmentów, przed właściwym segmentowaniem przetwarzanego tekstu, został przedstawiony w algorytmie 2.

Na początku słownik symboli inicjalizujemy pojedynczymi znakami powstałymi przez rozbicie poszczególnych słów wykorzystanego tekstu (algorytm 2 linia 4). Każdy wyraz jest odtąd reprezentowany jako sekwencja znaków zakończona dodatkowym symbolem końca słowa (na przykład '$\cdot$'). Zapobiega to niewłaściwemu łączeniu w pary rzekomo sąsiadujących znaków (nie występujących obok siebie w obrębie danego słowa) oraz umożliwia odtworzenie całych wyrazów po przeprowadzeniu tłumaczenia. Następnie zliczane są wszystkie następujące po sobie pary znaków (znaki w rodzaju $'',' E',' M'$) i zostaje wybrana ta pojawiająca się w zbiorze tekstowym najczęściej (algorytm 2 linia 5). Dalej (algorytm 2 linia 6) zastępujemy oba znalezione znaki przez nowy symbol utworzony z ich połączenia ($\{'',' E',' M'\} \rightarrow \{'',' EM'\}$). Każde następne połączenie skutkuje dołączeniem kolejnego symbolu do słownika symboli. Ostatecznie rozmiar słownika po dodaniu wszystkich symboli jest taki jak jego rozmiar pierwotny, powiększony o utworzone połączenia. Żeby poprawić wydajność całego algorytmu, można każdemu wyrazowi (symbolowi) przyporządkować wagę oznaczającą liczbę jego wystąpień w przetwarzanym tekście i z jej pomocą utrzymywać uporządkowany słownik.

Algorytm 2: Learn BPE (na podstawie[26])

Input: Ztokenizowany tekst z dopisaną liczbą wystąpień poszczególnych słów
Output: Najczęściej powtarzający się symbol
Data: Ztokenizowany tekst *inData*

$symDict \leftarrow inData$
$i \leftarrow 0$
while $i < liczba_iteracji$ **do**
    $symDict \leftarrow dodajSymboleDoSownika(symDict)$
    $mostFreq \leftarrow policzWystpieniaPar(symDict)$
    $symDict \leftarrow aktualizujSownik(symDict, mostFreq)$
    $i \leftarrow i + 1$

Implementacja

W realizowanym projekcie do przeprowadzenia operacji BPE skorzystano ze skryptu dostarczonego wraz z pakietem Marian. Jest on implementacją przedstawionego powyżej algorytmu (algorytm 2) uwzględniającą pewne dodatkowe modyfikacje. Na początku warto zauważyć, że w przeciwieństwie do oryginalnej idei BPE zastosowanie skryptu nie prowadzi do kompresji czystego tekstu źródłowego. Jego stosowanie może jednak prowadzić do zmniejszenia słownika zbudowanego na przetwarzanym tekście (ang. *vocabulary*) do ograniczonego zbioru symboli składających się na przetwarzany tekst. Liczba tokenów składających się na tekst zwiększa się wtedy jedynie w niewielkim stopniu.

Uruchomienie skryptów trenujących i stosujących BPE na korpusach (odpowiednio *learn_bpe.py* oraz *apply_bpe.py*) zawiera skrypt przedstawiony w kodzie źródłowym 3.16.

Kod źródłowy 3.16. Trenowanie i stosowanie BPE

```
47 ### Train BPE:
48 echo "training $folderName BPE..."
49 cat $corpseDir/corpus.text.tc.$src $corpseDir/corpus.text.tc.$tgt \
50 | $subwordNMT/learn_bpe.py -s 85000 \
51 > $corpseDir/deen.bpe
52 echo "done"
53
54 ### Corpus reparation
55 echo -e "Apply BPE to $folderName corpus:"
56 for lang in $src $tgt; do
57     echo "processing .$lang..."
58     cat $corpseDir/corpus.text.tc.$lang \
59     | $subwordNMT/apply_bpe.py -c $corpseDir/deen.bpe \
60     > $corpseDir/corpus.bpe.$lang
```

```
61      echo "corpus.bpe.$lang created"
62  done
63  echo "Apply BPE to $folderName validSet:"
64  for lang in $src $tgt; do
65      echo "processing .$lang..."
66      cat $corpseDir/validSet.text.tc.$lang \
67      | $subwordNMT/apply_bpe.py -c $corpseDir/deen.bpe \
68      > $corpseDir/validSet.bpe.$lang
69      echo "validSet.bpe.$lang created"
70  done
```

Pierwsza część przetwarzania to trenowanie modelu BPE (linie 49-51 kodu 3.16). Za pomocą polecenia *cat* (wiersz 49) zawartość korpusów *corpus.text.tc.*$*src* i *corpus.text.tc.*$*tgt* (odpowiednio *src* - angielski i *tgt* - niemiecki) zostaje przekazana do skryptu trenującego *learn_bpe.py* (wiersz 50). Parametr $-s$ z przekazaną mu wartością oznacza liczbę nowych symboli dodanych do słownika. Wytrenowany model zostaje zapisany w pliku *deen.bpe* dzięki właściwości przekierowania strumienia konsoli (wiersz 51). Utworzony model można zastosować do przetwarzanych korpusów i zbiorów walidacyjnych (odpowiednio wiersze 56-62 oraz 64-70).

Rozdział 4

Tagging

Tłumaczenie (zarówno "klasyczne" — przeprowadzane przez człowieka, jak i maszynowe — wykonywane przez wyspecjalizowane narzędzie komputerowe) wymaga znajomości słów języka docelowego. Można również skorzystać z dodatkowych informacji dostarczanych przez gramatykę danego języka. W przypadku tłumaczenia maszynowego trzeba więc dostarczyć modelowi tłumacza potrzebnych cech językowych zapisanych w odpowiedniej formie. Dostarczany jest więc precyzyjnie ustalony zbiór znaczników określających cechy językowe właściwe dla poszczególnych słów tekstu źródłowego. Można dzięki temu bardziej polegać na informacjach dotyczących małego odcinka korpusu.

Tagowanie ma również zaletę związaną z ograniczeniami sprzętowymi maszyny realizującej trenowanie tłumacza i tłumaczenie. Może pomóc w ograniczeniu konieczności (czasem pomaga uniknąć prawie całkowicie) korzystania z pamięci długotrwałej komputera.

4.1. Oznaczanie części mowy

Oznaczanie części mowy (ang. *Part Of Speech tagging*, w skrócie *POS tagging*) jest zadaniem polegającym na przyporządkowywaniu etykiet, które odpowiadają częściom mowy, do słów albo tokenów ([10]). Do przeprowadzenia tego procesu należy wybrać odpowiednie narzędzia, jak stosowany w tym wypadku *TreeTagger* (używany również do tokenizacji — patrz rozdział 3.2). Kolejną istotną kwestią jest wybór zestawu znaczników przyporządkowywanych słowom w procesie POS taggingu (tutaj *Penn Treebank tagset* opisany w rozdziale 4.2).

Termin tag oznacza etykietę przypisywaną każdemu słowu/tokenowi w korpusie tekstowym. Reprezentuje on część mowy oraz inne gramatyczne kategorie takie jak na przykład

TEXT: Is he doing it right?

TAGS: Is VBZ he PP doing VBG it PP right RB ? SENT

Rys. 4.1. Tekst z tagami - przykład

przypadek, czas, liczba (pojedyncza/mnoga) (*Sketch Engine* [4]). Wszystkie potrzebne cechy właściwe dla danego języka muszą wobec tego zostać w systematyczny sposób opisane i zapisane w zwięzłej formie. Zbiór etykiet może zostać ustalony na różnym poziomie szczegółowości. W podstawowej formie w tagach mogą zostać rozróżnione zaledwie główne części mowy (V - czasownik, N - rzeczownik, A - przymiotnik itd.). Można jednak pójść dalej i rozróżniać tagi na podstawie większej liczby cech gramatycznych. Mogą to być na przykład czasy, liczba (pojedyncza/mnoga), strona, wydźwięk, koniugacja. W zależności od potrzeb dopuszczalne jest też oczywiście dodanie dowolnego rodzaju elementów do tagów.

P1.a: Students do not sleep very much.
P1.b: Lack of sleep is harmful for people.
P2.a: Many sleepless nights can effect in dead appearance.
P2.b: turns too many sleepless nights into the dead.

Rys. 4.2. Kilka znaczeń wyrazów

Do czego mogą zatem służyć tagi wydobyte z tekstu? Reprezentują cechy gramatyczne poszczególnych słów w przetwarzanym tekście. Umożliwiają zatem uwzględnienie kryteriów językowych w tłumaczeniu maszynowym. Dla wielu języków to samo słowo można interpretować (w interpretacji gramatycznej) na kilka różnych sposobów. Przykładowo w przypadku języka angielskiego słowo *sleep* może oznaczać zarówno czasownik (ang. *verb*) "spać" — Rys. 4.2 P1.a jak i rzeczownik (ang. *noun*) "sen" — Rys. 4.2 P1.b. Podobnie słowo *dead* występuje jako przymiotnik (ang. *adjective*) "martwy" — Rys. 4.2 P2.a oraz jako rzeczownik (ang. *noun*) "zmarli" — Rys. 4.2 P2.b.

Istnieje również możliwość wykorzystania zestawu samych tagów bez słów, na podstawie których je wyodrębniono. Można odkrywać wzorce/konstrukcje gramatyczne bez odwoływania do samych wyrazów. Zapiszmy na przykład instrukcję:
•wyszukaj angielski rzeczownik w liczbie mnogiej niepoprzedzony przedimkiem (*article*).
Oczywiście możliwe jest również połączenie reguł analizy tekstu, przykładowo:
•wyszukaj przymiotnik "dead" występujący w tekście jako przymiotnik (*adjective*), po którym występuje rzeczownik (*noun*) w liczbie mnogiej.

Zbiór wszystkich POS-tagów spośród których wybiera się tagi charakteryzujące poszczególne słowa w przetwarzanym korpusie, określa się mianem *tagsetu*. Może mieć on różną postać (np. różną liczność) w zależności od opisywanego języka. Dla podobnych języków również tagsety mogą być podobne, jednak gdy języki są całkowicie niezwiązane ze sobą również odpowiadające im tagsety będą całkowicie różne.

4.1.1. Idea tagowania

Dysponując określonym tagsetem można zrealizować proces tagowania. Podzielmy podejście do tej kwestii na dwa główne sposoby:

- manualne (przez człowieka),
- automatyczne (przez narzędzia programistyczne).

Obecnie rzadko stosuje się metodę manualnego przypisywania znaczników/tagów (Sketch Engine [4]). Dopasowanie odpowiednich etykiet do wszystkich dostarczonych słów zajęłoby człowiekowi dużo czasu. Ponieważ jest to bardzo pracochłonny proces, aby osiągnąć zadowalające wyniki w rozsądnym czasie należałoby do tego zatrudnić wiele osób (wiąże się to też ze zwiększonymi kosztami). Należałoby również zwrócić uwagę na kwestię uzgadniania wyników tagowania. Obecnie uproszczono to zadanie przez zastosowanie wyspecjalizowanych narzędzi komputerowych. Nie przypisują one etykiet(POS tagów) do słów ale sprawdzają, czy nie występują niespójności między tagami przydzielonymi przez różnych autorów. Kiedy program wykryje różnice pomiędzy dostarczonymi wersjami etykietowania/tagowania danego słowa, anotatorzy muszą uzgodnić jak ma wyglądać dyskusyjny tag/etykieta. Mogą też dodać nowy znacznik do tagsetu, żeby przystosować go do nowych okoliczności.

Do manualnego tagowania ucieka się aktualnie jedynie w przypadku bardzo małych korpusów (takie podejście stosowane jest zazwyczaj w przypadku danych trenujących dla nowo tworzonych automatycznych POS taggerów). Ręczne tagowanie dużych korpusów (wiele miliardów słów) jest mało wydajne — w ich przypadku robi się użytek z narzędzi w pełni (lub przynajmniej w zasadniczej części) automatycznych. Narzędzia stosowane do etykietowania zazwyczaj określa się właśnie mianem *POS tagger* albo po prostu *tagger*. Poprawność działania wielu z dostępnych taggerów osiąga dokładność powyżej 98% (*Statistical Machine Translation* [15]).

Pojawiające się niedokładności wynikają często z drobnych błędów znajdujących się w przetwarzanym korpusie, na przykład takich jak "literówki" (np. *graet* zamiast *great*), albo użyte czasem wykrzyknienia przekształcające podstawowe słowo (np. *suuuuper*). Kolejny problem mogą stanowić pojawiające się w korpusie dwuznaczności. Ciężko z całą pewnością oszacować dla nich właściwą interpretację danego słowa. Ilustruje to przykład zdania: "Time flies." (*Sketch Engine* [4]). Rozumowanie może doprowadzić do wniosku, że wyrażenie składa się z rzeczownika i czasownika (noun + verb) — angielski idiom tłumaczony na: "Czas leci.". Z drugiej strony może ono też być następującymi po sobie czasownikiem i rzeczownikiem (verb + noun) — czyli "Zmierz (czas) much.".
Szczęśliwie (dla użytkowników) algorytmy zaimplementowane w powszechnie wykorzystywanych narzędziach znajdują rozwiązania powyższych trudności nie powodując powstawania kolejnych.

Przed wykonaniem samego tagowania korpusu potrzeba jednak wytrenować narzędzie. Używając jedynie małego zbioru (1 milion słów często wystarcza) manualnie przygotowanych danych treningowych tagger "uczy się" w jaki sposób etykietować słowa korpusu we właściwym procesie tagowania uwzględniając słowa w pewnym kontekście (na przykład po dwa słowa występujące przed i po danym słowie w zdaniu).
Oczywistym jest, że jakość działania taggera zależy od jakości danych treningowych — jeśli w dostarczonym zbiorze tekstowym występują przedstawione wyżej niedokładności, wpłynie to na wynik działania taggera. Podobna sytuacja miałaby miejsce w przypadku, gdyby w momencie wystąpienia niejednoznaczności w interpretacji etykietowania danych słów nie uzgodniono właściwego doboru tagów.

Niektóre zadania wymagają tagowania korpusów różnych języków. Można to rozwiązać na dwa sposoby:.

1. Pewne taggery operują na pojedynczych językach. Narzędzia te dysponują często zbiorem reguł i algorytmów wyspecjalizowanych pod kątem konkretnych dialektów. Mogą stosować różne charakterystyczne dla nich podejścia do analizy tekstu. Tagowanie korpusów w różnych językach należy wtedy realizować różnymi narzędziami przystosowanymi właśnie do tych tekstów.

2. Dostępne są narzędzia przystosowane do przetwarzania więcej niż jednego języka. Zasadnicze elementy taggerów pozostają wtedy wspólne pomiędzy dialektami. Różnicę stanowią jedynie osobne modele językowe dla każdego z języków.

4.2. Zestaw tagów Penn Treebank

W projekcie do realizacji tagowania posłużono się angielskim zestawem *Penn Treebank tagset*. Tabela 4.1 zawiera wszystkie tagi występujące w tym zbiorze wraz z prostym wyjaśnieniem samego symbolu (czasami rozwinięcie skrótu) oraz przykładami słów do których mogą go stosować. Całość pochodzi z materiałów udostępnionych przez *Sketch Engine*[3].

Tab. 4.1. Zestaw tagów Penn Treebank [3]

POS Tag	Description	Example
CC	coordinating conjunction	and
CD	cardinal number	1, third
DT	determiner	the
EX	existential there	there is
FW	foreign word	les
IN	preposition, subordinating conjunction	in, of, like
IN/that	that as subordinator	that
JJ	adjective	green
JJR	adjective, comparative	greener
JJS	adjective, superlative	greenest
LS	list marker	1)
MD	modal	could, will
NN	noun, singular or mass	table
NNS	noun plural	tables
NP	proper noun, singular	John
NPS	proper noun, plural	Vikings
PDT	predeterminer	both the boys
POS	possessive ending	friend's
PP	personal pronoun	I, he, it
PP$	possessive pronoun	my, his
RB	adverb	however, usually, naturally, here, good
RBR	adverb, comparative	better
RBS	adverb, superlative	best
RP	particle	give up
SENT	Sentence-break punctuation	. ! ?
		Ciąg dalszy nastąpi...

Tab. 4.1 Penn Tagset — kontynuacja

POS Tag	Description	Example
SYM	Symbol	/ [= *
TO	infinitive 'to'	togo
UH	interjection	uhhuhhuhh
VB	verb be, base form	be
VBD	verb be, past tense	was, were
VBG	verb be, gerund/present participle	being
VBN	verb be, past participle	been
VBP	verb be, sing. present, non-3d	am, are
VBZ	verb be, 3rd person sing. present	is
VH	verb have, base form	have
VHD	verb have, past tense	had
VHG	verb have, gerund/present participle	having
VHN	verb have, past participle	had
VHP	verb have, sing. present, non-3d	have
VHZ	verb have, 3rd person sing. present	has
VV	verb, base form	take
VVD	verb, past tense	took
VVG	verb, gerund/present participle	taking
VVN	verb, past participle	taken
VVP	verb, sing. present, non-3d	take
VVZ	verb, 3rd person sing. present	takes
WDT	wh-determiner	which
WP	wh-pronoun	who, what
WP$	possessive wh-pronoun	whose
WRB	wh-abverb	where, when
#	#	#
$	$	$
“	Quotation marks	‘ “
“	Opening quotation marks	‘ “
(	Opening brackets	({
)	Closing brackets	) }

Ciąg dalszy nastąpi...

Tab. 4.1 Penn Tagset — kontynuacja

POS Tag	Description	Example
,	Comma	,
:	Punctuation	– ; : — ...

Głównym powodem wyboru zestawu *Penn Treebank tagset* była znaczna liczba wyodrębnionych różnych etykiet słów (w powyższej tabeli zebrano 58 różnych tagów).

4.2.1. Objaśnienie tagów

W tabeli 4.1 podano jedynie podstawowe informacje o poszczególnych tagach. Niniejsza sekcja służy bardziej szczegółowemu przedstawieniu niektórych z nich (na podstawie artykułu Santorini[22] i danych ze Sketch Engine[3]).

ADJECTIVE — tag przymiotnika i jego odmiany

JJ (adjective) — przymiotnik

Przykłady: *simple, tired, nice*

Również związki wyrazów rozdzielonych łącznikiem "-" są traktowane jak przymiotniki **JJ**.

Przykłady: *one-of-a-kind, out-of-order*

JJR (adjective, comparative) — przymiotnik, porównawczy

Do tej grupy zaliczamy przymiotniki kończące się przyrostkiem porównawczym *-er*.

Przykłady: *bigger, smaller*

Wyrazy *less* i *more* używane jako przymiotniki występujące również są tagowane jako **JJR** (na przykład w stwierdzeniu *more or less spam*). Tak samo mogą być tagowane również w przypadku gdy występują samodzielnie.

Niektóre przymiotniki o znaczeniu porównawczym (np. *inferior*) nie kończą się przyrostkiem *-er*. Powinny zostać sklasyfikowane jako **JJ**. Podobnie przymiotniki kończące się przyrostkiem *-er* ale nie mające znaczenia porównawczego również powinny otrzymać etykietę **JJ**.

JJS (adjective, superlative) — przymiotnik, superlatyw

Tag ten przypisywany jest do słów kończących się na *-est* (także *-ost*).

Przykłady: *biggest, greatest, , worst*

Wyrazy *least* i *most* używane jako przymiotniki również są tagowane jako **JJS** (na przykład w stwierdzeniu *most or the least spam*). Tak samo mogą być tagowane również w przypadku gdy występują samodzielnie.

Niektóre przymiotniki o znaczeniu superlatywnym (np. *first, last*) nie kończą się przyrostkiem *-est*. Powinny zostać sklasyfikowane jako **JJ**.

NOUN — tag rzeczownika i jego odmiany

NN (noun, singular or mass) — rzeczownik, liczba pojedyncza albo grupa

NNS (noun plural) — rzeczownik w liczbie mnogiej

NP (proper noun, singular) — rzeczownik właściwy, liczba pojedyncza

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez przekształcenie tagu NNP z tagsetu Penn w wersji podstawowej.

NPS (proper noun, plural) — rzeczownik właściwy, liczba mnoga

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez przekształcenie tagu NNPS z tagsetu Penn w wersji podstawowej.

VERB — tag czasownika i jego odmiany

VV (verb, base form) — czasownik, forma podstawowa

VVD (verb, past tense) — czasownik, czas przeszły

VVG (verb, gerund/present participle) — czasownik, rzeczownik odczasownikowy/imiesłów czasu teraźniejszego

VVN (verb, past participle) — czasownik, imiesłów czasu przeszłego

VVP (verb, sing. present, non-3d) — czasownik, liczba pojedyncza czsu teraźniejszego, nie w trzeciej osobie

VVZ (verb, 3d person sing. present) — czasownik, trzecia osoba liczba pojedyncza czas teraźniejszy

VB (verb be, base form) — czasownik *be*, forma podstawowa

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *be* (być) od pozostałych nie modalnych (**VV**).

VBD (verb be, past tense) — czasownik *be*, czas przeszły

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *be* (być) od pozostałych nie modalnych (**VVD**).

VBG (verb be, gerund/present participle) — czasownik *be*, rzeczownik odczasownikowy/imiesłów czasu teraźniejszego

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *be* (być) od pozostałych nie modalnych (**VVG**).

VBN (verb be, past participle) — czasownik *be*, imiesłów czasu przeszłego

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *be* (być) od pozostałych nie modalnych (**VVN**).

VBP (verb be, sing. present, non-3d) — czasownik *be*, liczba pojedyncza czasu teraźniejszego, nie w trzeciej osobie

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *be* (być) od pozostałych nie modalnych (**VVP**).

VBZ (verb be, 3rd person sing. present) — czasownik *be*, trzecia osoba liczba pojedyncza czas teraźniejszy

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *be* (być) od pozostałych nie modalnych (**VVZ**).

VH (verb have, base form) — czasownik *have*, forma podstawowa

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *have* (mieć) od pozostałych nie modalnych (**VV**).

VHD (verb have, past tense) — czasownik *have*, czas przeszły

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *have* (mieć) od pozostałych nie modalnych (**VVD**).

VHG (verb have, gerund/present participle) — czasownik *have*, rzeczownik odczasownikowy/imiesłów czasu teraźniejszego

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *have* (mieć) od pozostałych nie modalnych (**VVG**).

VHN (verb have, past participle) — czasownik *have*, imiesłów czasu przeszłego

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *have* (mieć) od pozostałych nie modalnych (**VVN**).

VHP (verb have, sing. present, non-3d) — czasownik *have*, liczba pojedyncza czasu teraźniejszego, nie w trzeciej osobie

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *have* (mieć) od pozostałych nie modalnych (**VVP**).

VHZ (verb have, 3rd person sing. present) — czasownik *have*, trzecia osoba liczba pojedyncza czas teraźniejszy

Tag powstał dzięki modyfikacji wprowadzonej dla TreeTaggera (rozdział 2.2) przez odróżnienie czasownika *have* (mieć) od pozostałych nie modalnych (**VVZ**).

Pozostałe tagi

CC (coordinating conjunction) — spójnik nadrzędny (koordynujący)

Przykłady: *and, but, or, nor*

Kategoria ta obejmuje także słowa takie jak *yet* (na przykład *It's old yet still servicable*). Do grupy zaliczają się też operatory matematyczne jak *plus, minus, times, over, less* (*times* w znaczeniu "pomnożone przez" albo "razy"; *over* w znaczeniu "podzielony przez"). Również *for* w znaczeniu *beacause* (ponieważ) można traktować jako element grupy **CC**.

CD (cardinal number) — liczebnik główny

DT (determiner) — określnik (determinator)

Fo tej grupy zaliczamy przedimki w rodzaju *a, an, the, no, every.* Mieszczą się w niej również nieokreślone determinatory (przedimki) jak *any, each, either, neither, that, this, these, those, another, some.* Wystąpienia słów *both, all* (na przykład w *both cases* albo *all papers*) są również tagowane jako **DT**, jeśli nie poprzedzają innego określnika albo zaimka dzierżawczego.

W wypadku, w którym za takim słowem (*both* albo *all*) występuje jednak zaimek dzierżawczy albo określnik taguje się je jako **PDT**.

Z kolei *such* jako że może występować z określnikiem (np. *no such idea*) powinien być traktowany jak przymiotnik (**JJ**). Jeśli jednak poprzedza określnik (*such a lovely view*) powinien otrzymać etykietę **PDT**.

EX (existential there) — *there* egzystencjalne

Jest to nieakcentowane *there* które powoduje przesunięcie podmiotu logicznego zdania i odmienionego czasownika.

Przykład: *There was a sunrise.*

FW (foreign word) — słowo obcojęzyczne

Słowo w innym (patrzymy z perspektywy języka angielskiego) języku. Na przykład wszystkie słowa stwierdzenia *Ad rem* albo *Ad maiorem Dei gloriam* powinny być tagowane jako **FW**

IN (preposition, subordinating conjunction) — przyimek, spójnik podrzędny

W tagsecie w wyraźny sposób nie rozróżnia się przyimka od spójnika podrzędnego. Jest to uwzględniane tylko w algorytmie przydzielania tagów.

IN/that (that as subordinator) — *that* jako spójnik podrzędny

LS (list marker) — znacznik listy

Cyfry (liczby, liczebniki), litery w formie używanej jako znaczniki kolejnych (pod)elementów w listach.

Przykład: *5. ; A.*

MD (modal) — czasownik modalny

Do tej grupy zaliczamy czasowniki, które w trzeciej osobie liczby pojedynczej nie są zakończone przyrostkiem *-s*.

Przykłady: *can, may, might, must, shall, will*

PDT (predeterminer) — (przed)określnik

Pojawiają się tutaj wyrazy podobne do słów o kategorii **DT** ale dotyczą słów, po których występują przedimki albo zaimki dzierżawcze.

Przykłady: *half this time, all these people*

POS (possessive ending) — końcówka dzierżawcza

Należą do tej grupy zakończenia słów jak *'s* albo samo *'*. Na etapie tokenizacji są one oddzielane od słów z którym są wcześniej złączone tworząc jakby osobny wyraz.

Przykłady: *John's baptism, The Elders' wisdom*

PP (personal pronoun) — zaimek osobowy

Należą do tej grupy właściwe zaimki osobowe jak na przykład *I, me, you, she, he, him* (wielkość liter nie ma tutaj znaczenia). Do tej kategorii zalicza się również nominalne zaimki dzierżawcze (jak *mine, yours, hers, his, ours, theirs*) a także zaimki zwrotne zakończone przyrostkiem *-self* lub *-selves*.

Formy przymiotnikowe zaimka dzierżawczego (np. *my, your, her, his, its, our, their*) mogą być również tagowane jako **PP$**.

PP$ (possessive pronoun) — zaimek dzierżawczy

Do tej grupy należą *my, your, her, his, its, our, their, one's*. Nominalne zaimki dzierżawcze (w rodzaju *mine, yours, hers, his, ours, theirs*) zalicza się do grupy **PP**.

RB (adverb) — przysłówek

Do tej grupy zazwyczaj należą słowa kończące się na *-ly*. Mieszczą się tu jednak też wyrazy określające stopień (np. *too, very, quite*), jak również wyrazy takie jak *indeed, enough* pisane na końcu zdania. Do grupy przysłówków **RB** należą też zaprzeczenia *not, never* i zakończenia słów *n't*. Przykład: *She is a friendly person indeed, very interesting case, can't do it, never again*

RBR (adverb, comparative) — przysłówek porównawczy

Do tej grupy należą słowa kończące się na *-er* i mające porównawcze znaczenie.

Przykład: *I have a better idea*

Przysłówki posiadające wymaganą końcówkę (*-er*) ale nie mające wyraźnie porównawczego znaczenia (np. *do it later*) należą do grupy **RB**.

RBS (adverb, superlative) — przysłówek, superlatyw

Do tej grupy należą przysłówki w znaczeniu superlatywnym (często kończące się przyrostkiem *-est*).

Przykłady: *lightest*

Przysłówek *most* w połączeniu *most every-* zaliczany jest jednak do grupy **RB**.

RP (particle) — partykuła

Do tej grupy należy wiele wyrazów monosylabicznych. Mogą one często pełnić też funkcję przysłówków (**RB**) i przyimków (**IN**).

SENT (sentence-break punctuation) — znak interpunkcyjny rozdzielający zdania

Do tej grupy należą znaki interpunkcyjne kończące (rozdzielające) zdania.

Przykład: *. ! ?*

SYM (symbol) — symbol

Do tej grupy należą znaki i wyrażenia matematyczne, które nie są angielskimi słowami. Nazwy jednostek, nazwy związków chemicznych itp. powinny być tagowane jak rzeczowniki.

Przykład: */ [= **

TO (infinitive 'to') — bezokolicznik *to*

Do tej grupy zaliczane jest słowo *to* kiedy jest używane jako wykładnik bezokoliczni-

ka (ang. *infinitive marker*). Kiedy występuje w roli przyimka powinno być zaliczone do grupy **IN**.

UH (interjection) — wykrzyknienie

Zaliczają się do tej grupy wyrazy takie jak na przykład *see* (np.*See, it's working like this*), *my* (np. *My, what a lovely morning*). Również słowa *well, yes* czy też *uh* tagowane są tagiem **UH**.

WDT (wh-determiner) — determinator (określnik) *wh-*

Do tej grupy zaliczany jest wyraz *which*. Wyraz *that* również włącza się do tej grupy, jeśli jest użyty jako zaimek względny (relative pronoun).

WP (wh-pronoun) — zaimek *wh-*

Do grupy tej zaliczane są słowa takie jak *who, whom, what*.

WP$ (possessive wh-pronoun) — zaimek dzierżawczy *wh-*

Do tej grupy zalicza się słowo *whose*.

WRB (wh-adverb) — przysłówek *-wh*

Do tej grupy zaliczane są słowa takie jak na przykład *why where, how*. Tę kategorię można również przypisać do słowa *when* — jeśli występuje jako przysłówek czasu (temporal adverb) (Przykład 1).

Przykład 1: *When i saw him, he started running.* Jeśli jednak *when* przyjmuje w zdaniu znaczenie *if* (Przykład 2) powinien być traktowany jak spójnik podrzędny (i być tagowany jako **IN**).

Przykład 2: *I appreciate it, when you try to do something useful.*

4.3. Proces tagowania

Zazwyczaj proces tagowania opisuje się jako rodzaj sekwencji: tag pierwszego z rozważanych słów może zależeć od tagu słowa drugiego czy trzeciego, tag słowa drugiego może zależeć od słowa pierwszego, trzeciego i czwartego i tak dalej (Goldberg [12]). Można to z powodzeniem przybliżyć użyciem "okna" o wielkości dwóch słów przykładanego do analizowanego słowa z obu stron (przed i za słowem). Jeśli tagowanie zachodzi w ściśle ustalonym porządku (na przykład od lewej do prawej strony) przy ustalaniu kolejnych tagów można się posłużyć etykietami zatwierdzonymi wcześniej (np. pochodzącymi od słów po lewej stronie). Przed posłużeniem uzgodnionymi wcześniej informacjami trzeba określić ich źródło. Informacje o tagach opiera się na dwóch typach przesłanek:

- przesłanki wewnętrzne,
- przesłanki zewnętrzne.

Przesłanki wewnętrzne wiążą się z konkretnym słowem jako takim. Należą do nich czynniki takie jak choćby tożsamość słowa (ang. *the identify of the word*) — dla przykładu, niektóre mogą z większym prawdopodobieństwem być rzeczownikami niż czasownikami (według danych z Google Translate `https://translate.google.com/` taka sytuacja zachodzi choćby dla przywołanego wcześniej angielskiego słowa *time*).
Przesłankę wewnętrzną stanowią również przedrostki, przyrostki, czy ortograficzna forma słowa (ang. *orthographic shape of the word*). Na przykład w wielu językach słowa zaczynające się wielką literą najprawdopodobniej są nazwami własnymi. Jeśli chodzi o część mowy to na przykład angielskie słowo kończące się przyrostkiem *-ed* najpewniej jest czasownikiem w czasie przeszłym, czy też wyraz zaczynający się na *un-* jest najpewniej przymiotnikiem (zaprzeczeniem). Również liczba wystąpień słowa, należąca do grupy przesłanek wewnętrznych, może mieć znaczenie — rzeczowniki występują rzadziej niż inne części mowy.

Do przesłanek zewnętrznych zaliczają się (analogicznie do przesłanek wewnętrznych) nazwa słowa, ich przedrostki czy przyrostki. W tym wypadku dotyczą one jednak słów będących w sąsiedztwie tego aktualnie analizowanego. Również POS tagi słów poprzedzających (słów będących w obrębie okna branego pod uwagę) mają w tym wypadku znaczenie.

Rozdział 5

Projekt magisterski

Proces trenowania modelu tłumacza automatycznego można podzielić na kilka odrębnych etapów. Przybliżony porządek ich realizacji prezentuje Algorytm 3).

Algorytm 3: Trening

Input: Surowe dane tekstowe,
języki źródłowy i docelowy we wspólnych plikach
Output: Model tłumacza automatycznego
Data: Pliki tekstowe $A1, A2, A3, A4$

1 $cleanData = zbierz_i_wstepnie_przetworz(A1, A2, A3, A4)$
2 $prepreparedData \leftarrow taguj_i_tokenizuj(cleanData)$
3 $readyData \leftarrow wykonaj_ostatnie_szlify(prepreparedDat)$
4 $model \leftarrow trenuj(readyData)$

5.1. Gromadzenie i wstępne przetwarzanie

Działania reprezentowane w algorytmie 3 przez metodę *zbierz_i_wstepnie_przetworz* są realizowane w ramach przetwarzania wstępnego i należą do nich takie operacje jak:

- gromadzenie danych tekstowych (Rozdział 3.1.1),
- skracanie korpusów (Rozdział 3.1.2),
- czyszczenie korpusów (Rozdział 3.1.3).

5.1.1. Gromadzenie danych tekstowych

W ramach gromadzenia danych tekstowych po działaniach opisanych we wskazanym rozdziale (3.1.1) zachodzi jeszcze proces rozdzielania utworzonego pliku tekstowego na korpus

języka źródłowego (niemiecki) i korpus języka docelowego (angielski) — kod źródłowy 5.1. Od tego miejsca działania wykonywane na każdym z korpusów są (miejscami) uzależnione od jego języka (każde z nich dotyczy również korpusów walidacyjnych, zbiory testowe są przetwarzane tak, jak korpus języka źródłowego *de*).

Kod źródłowy 5.1. Rozdzielanie języków

```
42 ### Separate languages
43 echo "Separating languages..."
44 cut -f 1 $corpseDir/rawCorpse.shuf.tsv >
       $corpseDir/rawCorpse.shuf.$src
45 cut -f 2 $corpseDir/rawCorpse.shuf.tsv >
       $corpseDir/rawCorpse.shuf.$tgt
46 echo "done\n"
```

5.1.2. Czyszczenie korpusów

W ramach czyszczenia korpusów (m.in.) przed działaniami opisanymi we wskazanym rozdziale (3.1.3) zachodzi jeszcze proces normalizacji interpunkcji (kod źródłowy 5.2). Zastosowany skrypt *normalize – punctuation.perl*, należący do narzędzia *Moses*, poprawia znaki interpunkcyjne, aby reprezentowane były przez właściwe (dla danego języka) znaki w kodowaniu UTF.

Kod źródłowy 5.2. Normalizacja interpunkcji

```
59 ### Normalize punctuation
60 echo "Normalise punctuation"
61 for lang in $src $tgt; do
62     echo "Creating corpus.np.$lang..."
63     cat $corpseDir/corpse$folderName.$lang \
64     | $mosesScr/tokenizer/normalize-punctuation.perl -l $lang -
           threads 12 \
65     > $corpseDir/corpus.np.$lang
66     echo "done"
67     echo "Creating validSet.np.$lang..."
68     cat $corpseDir/validSet.$lang \
69     | $mosesScr/tokenizer/normalize-punctuation.perl -l $lang -
           threads 12 \
70     > $corpseDir/validSet.np.$lang
71     echo "done"
```

```
72 done
```

5.2. Tagowanie i tokenizacja

Zgromadzone, znormalizowane i wyczyszczone korpusy są przygotowane do przeprowadzenia tokenizacji i tagowania (w algorytmie 3 reprezentuje to operacja *taguj_i_tokenizuj*). Dla części niemieckiej (tokenizacja) proces ten został opisany w rozdziale 3.2. Proces przetwarzania części dotyczącej języka angielskiego (tagowanie z tokenizacją) jest opisany w podrozdziale 5.2.1.

5.2.1. Implementacja tagowania w projekcie

Wiele pakietów operujących na tekście umożliwia zrealizowanie kilku zadań związanych z jego przetwarzaniem. Taka sytuacja ma miejsce w przypadku narzędzia TreeTagger (podrozdział 2.2) stosowanego w niniejszym projekcie za pośrednictwem biblioteki *treetaggerwrapper* (podrozdział 2.2.4). Przeprowadza nie tylko tokenizację (co pokazano w podrozdziale 3.2.1) ale przede wszystkim służy do tagowania (na co wskazuje sama nazwa). W realizowanym projekcie tagowanie zastosowano dla tekstów w języku angielskim. Pełny skrypt implementujacy ten proces zawarto w dodatku B (B.1) a objaśnienia przedstawiono poniżej.

Kod źródłowy 5.3. Angielskie zbiory tekstowe

```
11 corpse_dir1 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.clean.en'
12 corpse_dir2 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.np.en'
13 corpse_dir = [corpse_dir1, corpse_dir2]
14
15 # tagged_dir = 'tmpTagged.en'
16 tagged_dir1 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.tg.en'
17 tagged_dir2 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.tg.en'
18 tagged_dir = [tagged_dir1, tagged_dir2]
```

W kodzie źródłowym 5.3 zebrane zostały zbiory tekstowe używane w dalszym przetwarzaniu. Zawarte są one w następujących plikach: korpus w języku angielskim (wiersz

11. kodu źródłowego 5.3) i angielska strona zbioru walidacyjnego (wiersz 12. kodu źródłowego 5.3) – ten ostatni służy jako zbiór kontrolny w dalszym procesie trenowania modelu tłumacza.

Kod źródłowy 5.4. Tagowanie – metoda *start_test*

```
21 def start_test(n=None):
```

Zasadnicza część przetwarzania zachodzi w metodzie *start_test* (kod 5.4).

Kod źródłowy 5.5. Tagowanie – importowanie bibliotek

```
23     import treetaggerpoll as ttp
24     import treetaggerwrapper as ttw
```

Po zaimportowaniu potrzebnych bibliotek (kod 5.5) dla każdego używanego zbioru faktyczne przetwarzanie zaczyna się od wiersza 32. (kod 5.6).

Kod źródłowy 5.6. Tagowanie – metoda *TaggerProcessPoll*

```
32         p = ttp.TaggerProcessPoll(workerscount=n, TAGLANG="en")
```

Dla każdego używanego zbioru powstaje tu sondaż procesów (ang. *poll*) dla wybranego języka ($TAGLANG =$ "*en*") i określonej liczby procesów ($workerscount = n$). Ich liczbę ustala się w miejscu wywołania metody *start_test* – domyślnie zmienna n przyjmuje tam wartość pustą *None* co w efekcie oznacza przyjęcie przez metodę *ProcessPoll* wartości zgodną z maksymalną liczbą wątków dostępnych dla procesora komputera na którym zachodzi całość przetwarzania. Jeśli chodzi o wybrany język, to używa się właściwego mu skrótu (występujący tutaj skrót *en* oznacza język angielski). Dalsze przetwarzanie można podzielić na dwa etapy:

1. tagowanie wszystkich wierszy przetwarzanego pliku tekstowego (kod źródłowy 5.7 i 5.8, wiersze od 35 do 43),

2. zapisanie sekwencji tokenów i tagów do pliku tekstowego (kod źródłowy 5.9, wiersze od 47 do 68).

Kod źródłowy 5.7. Tagowanie – operacja *open*

```
35         with open(cd, 'r') as in_file:
```

Podstawę dla wyszczególnionych etapów przetwarzania stanowi oczywiście tagowanie, realizowane w ramach pierwszego etapu. Na jego początku przetwarzany plik zostaje

otwarty "do odczytu" – odpowiedni deskryptor zostaje przypisany do zmiennej *in_file* (kod 5.7). Dla każdego zdania pliku (pętla *for* kod 5.8) zostaje następnie przeprowadzone tagowanie.

Kod źródłowy 5.8. Tagowanie – metoda *tag_text_async*

```
40              for sentence in in_file.readlines():
41                  # print('sen:'+sentence, end='')
42                  taggSet.append(p.tag_text_async(sentence,
                        notagurl=True,
43                                                  notagemail=True,
                                                        notagip=True,
                                                        notagdns=True,
                                                        nosgmlsplit=True))
```

W tym celu wykorzystana zostaje metoda *tag_text_async* opisana w rozdziale 2.2.4. Przyjmuje ona kilka parametrów, których wartości wpływają na użyteczność wyników jej działania (patrz tabela 2.2). Odmienne od domyślnych wartości użyte tutaj zebrano w tabeli 5.1.

Tab. 5.1. Parametry metody *tag_text_async* dla tagowania zbiorów

parametr	wartość
text	sentence
notagurl	True
notagemail	True
notagip	True
notagdns	True
nosgmlsplit	True

Znaczenie zmienionych parametrów jak opisano następująco:

text — przekazana została wartość zmiennej *sentence*,
czyli przetwarzane w danym momencie zdanie (wiersz pliku tekstowego);

notagurl — przekazana została wartość *True*,
dzięki temu wartości adresów url pozostaną niezmienione;

notagemail — przekazana została wartość *True*,
dzięki temu wartości adresów email pozostaną niezmienione;

notagip — przekazana została wartość *True*,
dzięki temu wartości adresów ip pozostaną niezmienione;

notagdns — przekazana została wartość *True*,
dzięki temu wartości nazw dns pozostaną niezmienione;

notagsgmlsplit — przekazana została wartość *True*,
dzięki temu unika się podziału na znaczniki sgml co pozwala zachować niezmieniony tekst.

Z uwagi na to, że w przypadku języka angielskiego tagowanie było wymagane (przeciwnie niż w dla języka niemieckiego — rozdział 3.2.1), nie zmieniano domyślnej wartości *False* dla parametru *prepronly*.

Przetwarzanie zdań powyższą metodą skutkuje otrzymaniem listy list tokenów przeplatanych tagami (lista zdań/linii złożonych z naprzemiennie występujących: poszczególnych słów i tagów opisujących dane słowo). Do dalszego przetwarzania potrzebny jest plik tekstowy o ustalonej składni. W tym miejscu należy zrealizować drugi z wyszczególnionych wcześniej etapów (Kod źródłowy 5.9).

Kod źródłowy 5.9. Tagowanie – wyodrębnianie tagów i zapis wyniku do plików

```
47          with open(tagged_dir[j], 'w') as out_file:
48              print(tagged_dir[j]+' opened to write')
49              print('Make tags & write file...')
50              for i, r in enumerate(taggSet):
51                  r.wait_finished()
52                  # print(i, str(r.result))
53                  currLine = currLine+1
54                  taggSen = ttw.make_tags(
55                      r.result, exclude_nottags=True,
                            allow_extra=True)
56                  for token in taggSen:
57                      # print(token[0]+' '+token[1]+' ', end='')
58                      out_file.write(token[0]+' '+token[1]+' ')
59                  # print()
60                  out_file.write('\n')
61
62                  if (currLine == PACK_SIZE):
63                      currLine = 0
64                      print('line tagged:', i+1, '\tfirst word:',
65                            taggSen[0][0], '\ttag:', taggSen[0][1])
66
67                  taggSet[i] = None
```

```
68                 print('Tagged', i+1, 'lines')
```

Kod źródłowy 5.9 przedstawia implementację zapisu zawartości utworzonej uprzednio listy do pliku tekstowego. Na początku (wiersz 47) zostaje otwarty plik "do zapisu" ("w") — $open(tagged_dir[j],'w')$, w zmiennej $tagged_dir$ zapisane są ścieżki dostępu wszystkich plików tworzonych na skutek działania pełnego skryptu a wartość $[j]$ oznacza wybór konkretnego z nich (oznaczonego indeksem j). Wartość zwracana przez funkcję *open* przypisywana jest do zmiennej out_file używanej w dalszych krokach. Dalej (od linii 50.) wyodrębniane są tagi we wszystkich zdaniach i następuje ich zapis (wraz z tagami) do pliku. Na początku dla poszczególnych wierszy tekstu przetwarzanie jest wstrzymane do momentu, w którym tagowanie zdania zostanie zakończone (tag_text_async działa asynchronicznie w różnych procesach), wykonuje to $r.wait_finished$ w linii 51.

Kod źródłowy 5.10. Tagowanie – wyodrębnianie wszystkich tagów

```
54                 taggSen = ttw.make_tags(
55                     r.result, exclude_nottags=True,
                           allow_extra=True)
```

Dalej z aktualnie przetwarzanego wiersza tekstu wyodrębniane są wszystkie tagi (patrz kod źródłowy 5.10). W przedstawionym kodzie *ttw* jest nazwą oznaczającą bibliotekę *treetaggerwrapper* z której pochodzi używana do tego funkcja *make_tags* (opisana w rozdziale 2.2.4). Jako wartość pierwszego parametru przyjmuje ona aktualnie analizowane zdanie ($r.result$). Parametr $exclude_nottags$ przyjmuje wartość $True$, co oznacza, że w wyniku przetwarzania nie będą się pojawiać znaczniki $NotTag$ zamiast jego fragmentów. Przypisanie $allow_extra = True$ gwarantuje, że dzięki temu wszystkie etykiety długich zdań również będą wyodrębnione.

Kod źródłowy 5.11. Tagowanie – zapis tekstu i tagów do pliku

```
54                 for token in taggSen:
55                     # print(token[0]+' '+token[1]+' ', end='')
56                     out_file.write(token[0]+' '+token[1]+' ')
57                 # print()
58                 out_file.write('\n')
```

Dalej rozpoczyna się pętla *for* (Kod źródłowy 5.11), wewnątrz której dokonany zostaje zapis do pliku poszczególnych słów i odpowiadających im wydzielonych wcześniej tagów. Metoda $write(...)$ zapisuje zawartość nawiasu (...) do pliku określonego zmienną out_file. Forma zapisywanego tekstu określona jest jako $token[0] + ' ' + token[1] + ' '$. Oznacza to

zapisywanie kolejno słowa ($token[0]$), spacji, tagu ($token[1]$) i spacji dla każdego słowa (tokenu) występującego w zdaniu. W ten sposób każdy wiersz w pliku docelowym składa się z naprzemiennie występujących słów i tagów, gdzie każde z nich (zarówno słowa i tagi) jest oddzielone spacją. Metoda $out_file.write('\backslash n')$ występująca zaraz po pętli powoduje dodanie znaku nowej lini po ukończeniu wpisywania tokenów przetwarzanego zdania.

Komentarza może wymagać pętla for w liniach 62-65 kodu źródłowego 5.12. Jest ona zaledwie formą informowania użytkownika o numerze zapisywanego w danym momencie wiersza. Stała $PACK_SIZE$ odpowiada za krok, z jakim wypisywane są w konsoli kolejne informacje o stanie przetwarzania (krok ten wyrażony jest w liczbie przetworzonych wierszy).

Kod źródłowy 5.12. Tagowanie – wyświetlanie komunikatów dot. przetwarzania

```
62                  if (currLine == PACK_SIZE):
63                      currLine = 0
64                      print('line tagged:', i+1, '\tfirst word:',
65                            taggSen[0][0], '\ttag:', taggSen[0][1])
```

Pozostała kwestia uruchomienia powyższego skryptu skryptu. Dokonano tego za pomocą skryptu konsoli 3.14 opisanego przy okazji omawiania procesu tokenizacji.

5.3. Ostatnie szlify

Kod źródłowy 5.13. Ostatnie szlify

```
97  ### Separating text and tags
98  echo "Separate text and tags"
99  chmod 777 $workingDir/separate-text-tags-$folderName.py
100 python3 $workingDir/separate-text-tags-$folderName.py #
        corpus.tag.$src & corpus.txt.$src
101
102 ### Training/truecasing an and training/applying bpe: corpses,
        validation set, in-s
103 chmod 777 $workingDir/true-bpe.sh
104 $workingDir/true-bpe.sh $folderName
105
106 ### Inserting taggs to text
107 chmod 777 $workingDir/insert-tags-$folderName.py
108 python3 $workingDir/insert-tags-$folderName.py
```

Przed rozpoczęciem trenowania modelu tłumacza z wykorzystaniem narzędzia Marian (2.1) należy wykonać ostatnie szlify (*wykonaj_ostatnie_szlify* w algorytmie 3) na używanych zbiorach tekstowych. Można wyszczególnić tutaj kolejne etapy rozdzielania tekstu i tagów (lnie 97-100 kodu 5.13 opisane dalej w podrozdziale 5.3.1), truecasing ze stosowaniem kodowania BPE (linie 102-104 kodu 5.13 opisane w rozdziale 3.3) i wstawianie tagów do tekstu (linie 106-108 kodu 5.13 opisane w rozdziale 5.3.2).

5.3.1. Rozdzielanie tekstu i tagów

Każdy plik z otagowanym korpusem (pliki angielskie) na potrzeby dalszego przetwarzania rozdzielany jest na plik zawierający sam tekst i plik zawierający tagi. Na początku gwarantowane są uprawnienia co najmniej wystarczające do uruchomienia rozdzielającego skryptu (linia 99 kodu 5.13). Zmienna $\$folderName$ przechowuje nazwę pliku zawierającego przetwarzany korpus. Odpowiada ona rozmiarowi korpusu (np. dla korpusu mającego 1 milion wierszy jest to $1M$). Dalej uruchamiany jest sam skrypt $separate_text_tags - \$folderName.py$, przedstawiony w kodzie źródłowym 5.14.

Kod źródłowy 5.14. Oddzelanie tagów od tekstu

skrypt $separate_text_tags - 1M.py$

```
1  # -*- coding: utf-8 -*-
2
3  import sys
4
5  print(sys.argv[0])
6
7  gonito_path = '/home/marek/master/gonito/wmt-2017/'
8  file_list = ['corpse/1M/corpus',
9               'corpse/1M/validSet'
10              ]
11
12 tag_set = ['CC', 'CD', 'DT', 'EX', 'FW', 'IN', 'IN/that', 'JJ',
       'JJR', 'JJS', 'LS', 'MD', 'NN', 'NNS', 'NP', 'NPS', 'PDT',
       'POS', 'PP', 'PP$',
13             'RB', 'RBR', 'RBS', 'RP', 'SENT', 'SYM', 'TO', 'UH',
                  'VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ', 'VH',
                  'VHD', 'VHG', 'VHN', 'VHP', 'VHZ',
14             'VV', 'VVD', 'VVG', 'VVN', 'VVP', 'VVZ', 'WDT', 'WP',
                  'WP$', 'WRB',
```

```
15              '#', '$', '"', '``', '\'\'', '(', ')', ',', ':']
16
17  for i, fil in enumerate(file_list):
18      fil_path = gonito_path+fil
19      print('Working on'+fil_path+'...')
20
21      targetTagFile = open(fil_path+'.tag.en', 'w')
22      targetTextFile = open(fil_path+'.text.en', 'w')
23      preOut = []
24      sentence = []
25      with open(fil_path+'.tg.en', 'r') as preFile:
26          # preOut = preFile.readlines()
27          # print(preOut[:1],'\n')
28          for line in preFile.readlines():
29              # print('line:'+line, end='')
30              sentence = list(enumerate(line.split()))
31              for j, token in sentence:
32                  # drugi pomysl
33                  if j % 2 == 0:
34                      # print(token, end=' ')
35                      targetTextFile.write(token+' ')
36                  else:
37                      # print(token, end=' ')
38                      targetTagFile.write(token+' ')
39
40                  # # pierwszy pomysl
41                  # if token in tag_set:
42                  #     # print(token, end=' ')
43                  #     targetTagFile.write(token+' ')
44                  # elif token != sentence[j-1][1]:
45                  #     # print(token, end=' ')
46                  #     # print(sentence[j][1], end=' ')
47                  #     targetTextFile.write(token+' ')
48
49              # print()
50              targetTagFile.write('\n')
51              targetTextFile.write('\n')
52
53      print('done')
```

```
54      targetTagFile.close()
55      targetTextFile.close()
```

5.3.2. Wstawianie tagów

Po przetworzeniu plików z zastosowaniem kodowania BPE na potrzeby dalszego przetwarzania należy pliki języka docelowego (angielski) połączyć z oddzielonymi uprzednio tagami. Na początku gwarantowane są uprawnienia co najmniej wystarczające do uruchomienia rozdzielającego skryptu (linia 107 kodu 5.13). Zmienna $\$folderName$ przechowuje nazwę pliku zawierającego przetwarzany korpus. Odpowiada ona rozmiarowi korpusu (np. dla korpusu mającego 1 milion wierszy jest to $1M$). Dalej uruchamiany jest (linia 108 kodu 5.13) sam skrtypt $insert_tags - \$folderName.py$ przedstawiony w kodzie źródłowym 5.15.

Kod źródłowy 5.15. Włączenie tagów do tekstu

skrypt $insert_tags - 1M.py$

```
1  # -*- coding: utf-8 -*-
2
3  import sys
4
5  print(sys.argv[0])
6
7  file_list = ['/home/marek/master/gonito/wmt-2017/corpse/1M/corpus',
8               '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet'
9               ]
10 text_ext = '.bpe.en'
11 tag_ext = '.tag.en'
12
13 print('Insert tags to english corpus and validation set')
14 for i, fil in enumerate(file_list):
15     text_path = fil+text_ext
16     tag_path = fil+tag_ext
17
18     textFile = open(text_path, 'r')
19     print('Opened ', text_path+' for reading')
20     tagFile = open(tag_path, 'r')
21     print('Opened '+tag_path+' for reading')
22     # preOut = []
```

```
23      # sentence = []
24
25      print('Building '+fil+'.mg.en...')
26      with open(fil+'.mg.en', 'w') as mergedFile:
27          # preOut = preFile.readlines()
28          # print(preOut[:1],'\n')
29          tagsFromFile = tagFile.readlines()
30          for txln, line in enumerate(textFile.readlines()):
31              sentence = list(line.split())
32              tagsList = list(tagsFromFile[txln].split())
33              tgn = 0
34              for token in sentence:
35                  mergedFile.write(token+' ')
36                  if not(token[-2:] == '@@'):
37                      mergedFile.write(tagsList[tgn]+' ')
38                      tgn += 1
39
40              mergedFile.write('\n')
41
42      print('done\n')
43      textFile.close()
44      tagFile.close()
```

5.4. Trening modelu tłumacza

Niniejszy rozdział opracowano z wykorzystaniem informacji z dokumentacji frameworka Marian [8]

Etapem, do którego miało doprowadzić przygotowanie korpusów i zbiorów walidacyjnych (zbiorów testowych potrzebnych są przy ocenie jakości tłumaczenia) jest trening modelu tłumacza automatycznego (reprezentowany w algorytmie 3 przez funkcję *trenuj*). Wykorzystany został do tego pakiet narzędzi Marian opisany w rozdziale 2.1. Kod źródłowy 5.16 przedstawia definicję wszystkich używanych w skrypcie zmiennych (linie 4-28), polecenie tworzenia pliku konfiguracyjnego (linie 32-52), oraz uruchomienie procesu trenowania modelu (linie 54-57).

Kod źródłowy 5.16. Trening modelu tłumacza

skrypt $train - s2s - 1M - tag.sh$

```
#!/bin/bash
echo "$0"

src=de
tgt=en
type=s2s

folderName=1M
corpseLen=1000000
maxSenLen=1024
validSetLen=3000

gonito=/home/marek/master/gonito
wmt2017=$gonito/wmt-2017
train=$wmt2017/train
testA=$wmt2017/test-A
dev0=$wmt2017/dev-0
dev1=$wmt2017/dev-1

workingDir=/home/marek/master/src
marianDir=/home/marek/master/marian-dev
mosesScr=$marianDir/src/examples/moses-scripts/scripts
subwordNMT=$marianDir/src/examples/subword-nmt/subword_nmt
corpseDir=$wmt2017/corpse/$folderName
modelDir=$wmt2017/model/$folderName-$type-tag
mtm18l=/home/marek/master/mtm2018labs
trueModel=/home/marek/master/corpse/wmt17/true
trueModelName=truecase-model

mkdir -p $modelDir

# first create config file
echo "Create config file"
marian \
--model $modelDir/model.tg.npz --type $type \
--train-set $corpseDir/corpus.bpe.$src $corpseDir/corpus.mg.$tgt \
--vocabs $modelDir/vocab.tg.$src.yml $modelDir/vocab.tg.$tgt.yml \
```

```
38 --mini-batch-fit --workspace 2500 \
39 --valid-mini-batch 1 \
40 --layer-normalization --exponential-smoothing \
41 --dropout-rnn 0.2 --dropout-src 0.1 --dropout-trg 0.1 \
42 --valid-metrics cross-entropy bleu \
43 --valid-set $corpseDir/validSet.bpe.$src
      $corpseDir/validSet.mg.$tgt \
44 --valid-freq 5000 \
45 --beam-size 12 --normalize 1 \
46 --early-stopping 5 \
47 --save-freq 5000 --overwrite --keep-best \
48 --log $modelDir/train.tg.log --valid-log $modelDir/valid.tg.log \
49 --disp-freq 1000 --quiet-translation \
50 --seed 1111 \
51 --dump-config > $modelDir/config.tg.yml
52 echo "config file created"
53
54 # Use the config file to start our training process
55 echo "Train with config file"
56 marian -c $modelDir/config.tg.yml
57 echo "Advanced model trained"
```

Znaczenie wszystkich używanych zmiennych jest identyczne jak w przypadku zmiennych o tak samo brzmiącej nazwie używanych w skryptach powłoki systemu linux (*.sh*), które były stosowane w celu przygotowania korpusu (i innych zbiorów tekstowych). Ścieżki dostępu reprezentowane przez większość ze zmiennych są specyficzne dla instancji systemu, na którym realizowane jest przetwarzanie.

Kod źródłowy 5.16 w liniach 34-51 przedstawia wywołanie procedury *marian* w celu utworzenia pliku konfiguracyjnego dla treningu modelu tłumacza automatycznego. Parametry zostały dobrane tak, by było możliwe uruchomienie treningu a trenowanie zakończyło się w rozsądnym czasie (max około jednej doby). Manualnie ustalono wartości następujących parametrów:

model (wiersz 35),

w pliku (o określonej ścieżce dostępu) zostanie zapisany utworzony model tłumacza automatycznego (utworzony model jest również kompatybilny z pakietem narzędzi Nematus[24]);

type (wiersz 35),

określa model sieci neuronowej jaki ma zostać zastosowany do treningu modelu tłumacza automatycznego (w tym projekcie jest to model $s2s$ dostarczony wraz z frameworkiem Marian);

train-set (wiersz 36),

ścieżki dostępu do korpusów treningowych podane wraz z nazwą. Pierwszy z podanych to korpus w języku źródłowym (niemiecki), drugi to korpus w języku docelowym (angielski);

vocabs (wiersz 37),

ścieżki dostępu, pod jakimi mają być zapisane słowniki dotyczące używanych języków. Pierwsza z podanych dotyczy języka źródłowego (niemiecki), druga dotyczy języka docelowego (angielski);

valid-metrics (wiersz 42),

metryki, w jakich obliczana będzie jakość modelu podczas walidacji;

valid-set (wiersz 43),

ścieżki dostępu do zbiorów walidacyjnych (podane wraz z nazwą). Pierwszy z podanych to zbiór walidacyjny w języku źródłowym (niemiecki), drugi to zbiór walidacyjny w języku docelowym (angielski);

valid-freq (wiersz 44),

częstość przeprowadzania walidacji wyrażona w liczbie iteracji;

early-stopping (wiersz 46),

określa "cierpliwość" (ang. *patience*) z jaką będą przeprowadzane kolejne walidacje. Wartość 5 oznacza, że trening zostanie zakończony, jeśli w 5 kolejnych walidacjach ocena pierwszej metryki określonej w $valid - metrics$ się nie poprawi;

log (wiersz 48),

w pliku określonym tym parametrem zapisywany jest log prezentujący przebieg trenowania modelu;

valid-log (wiersz 48),

w pliku określonym tym parametrem zapisywany jest log prezentujący przebieg walidacji modelu w trakcie trenowania;

dump-config (wiersz 51),

wyrzuca plik zdefiniowaną konfigurację w formacie YAML. Konstrukcja $dump - config > dir/config.yml$ powoduje zapisanie pliku konfiguracyjnego do pliku $config.yml$, którego adres jest określony ścieżką dir.

Po zapisaniu parametrów do pliku konfiguracyjnego następuje właściwe uruchomienie trenowania tłumacza automatycznego za pomocą Mariana. Dokonująca tego instrukcja znajduje się w lini 56 kodu źródłowego 5.16

5.5. Tłumaczenie i postprocessing

Niniejszy rozdział opracowano na podstawie fragmentu dokumentacji frameworka Marian [7]

Podstawą przeprowadzenia ewaluacji tłumaczeń i porównania wyników dostarczonych za pomocą różnych modeli tłumaczy automatycznych jest wykonanie tłumaczenia korpusów testowych. To zadanie zrealizowano za pomocą skryptu zamieszczonego w kodzie źródłowym 5.17.

Kod źródłowy 5.17. Tłumaczenie (model uwzględniający tagowanie)

skrypt $translate - 1M.tg.sh$

```
1  #!/bin/bash
2
3  src=de
4  tgt=en
5  type=s2s
6
7  folderName=1M
8  corpseLen=1000000
9  maxSenLen=1024
10 validSetLen=3000
11
12 gonito=/home/marek/master/gonito
13 wmt2017=$gonito/wmt-2017
14 train=$wmt2017/train
15 testA=$wmt2017/test-A
16 dev0=$wmt2017/dev-0
17 dev1=$wmt2017/dev-1
```

```
18
19 workingDir=/home/marek/master/src
20 marianDir=/home/marek/master/marian-dev
21 mosesScr=$marianDir/src/examples/moses-scripts/scripts
22 subwordNMT=$marianDir/src/examples/subword-nmt/subword_nmt
23 corpseDir=$wmt2017/corpse/$folderName
24 modelDir=$wmt2017/model/$folderName-$type-tag
25 mtm18l=/home/marek/master/mtm2018labs
26 trueModel=/home/marek/master/corpse/wmt17/true
27 trueModelName=truecase-model
28
29 # model=$1     # model.npz
30 # vocabSrc=$2 # vocab.de.yml
31 # vocabTgt=$3 # vocab.en.yml
32
33 # model=$modelDir/model.tg.npz
34 model=$modelDir/model.tg.npz.best-cross-entropy.npz
35 vocabSrc=$modelDir/vocab.tg.$src.yml
36 vocabTgt=$modelDir/vocab.tg.$tgt.yml
37
38 echo "Granting permissions"
39 for fil in $dev0 $dev1 $testA; do
40     # chmod 777 $fil/in-pre.tsv
41     chmod 777 $fil/in.bpe.tsv
42 done
43 chmod 777 $workingDir/translateWithMarian.sh
44 echo "Permissions 777 granted to in-pre.tsv in dev-0, dev-1,
    test-A and to translateWithMarian.sh\n"
45
46 echo "Translating corpuses dev-0, dev-1 and test-A\n"
47 for fil in $dev0 $dev1 $testA; do
48     echo "Processing $fil"
49     # $workingDir/translateWithMarian.sh $model $vocabSrc
        $vocabTgt < $fil/in-pre.tsv > $fil/out-pre.tg.tsv
50     $workingDir/translateWithMarian.sh $model $vocabSrc $vocabTgt <
         $fil/in.bpe.tsv > $fil/out-pre.tg.tsv
51     echo "done for $fil"
52     echo "created $fil/out-pre.tg.tsv\n"
53 done
```

```
54 echo "Translation finished\n"
```

Podonie jak w skryptach uruchamiających procesy przetwarzania wstępnego albo jak w przypadku skryptu odpowiedzialnego za inicjalizację treningu modelu tłumacza automatycznego w kodzie źródłowym 5.17 w wierszach do lini 27 ustalane są wartości używanych zmiennych. W liniach 34-36 nazwy plików (wraz ze ścieżkami dostępu) zawierających kolejno używany model tłumacza automatycznego i utworzone w trakcie treningu słowniki: język źródłowy i docelowy. Każda z nich (ścieżek z nazwami pliku) zostaje przypisana do zmiennych, odpowiednio *model*, *vocabSrc* i *vocabTgt*.

Wywoływanie translatora zamieszczono w pętli *for* znajdującej się w liniach 47-53 skryptu $translate-1M.tg.sh$. Wykorzystywana w jej obrębie zmienna *fil* przyjmuje podczas każdego przebiegu pętli inną wartość równą zawartości zmiennych kolejno: $dev0$, $dev1$ i $testA$. Są to ścieżki dostępu do kolejnych (trzech) zbiorów testowych wytrenowanych razem ze stroną źródłową korpusu treningowego. Samo wywołanie skryptu uruchamiającego translatora (kod 5.18) znajduje się w lini 50. Jako parametry przyjmuje on zdefiniowane wcześniej ścieżki dostępu do modelu tłumacza i słowników w języku źródłowym i docelowym. Sam tekst do przetłumaczenia przekazywany jest do tego skryptu za pomocą potoku systemu Linux.

Kod źródłowy 5.18. Tłumaczenie (Skrypt uruchamiający tłumaczenie)

skrypt $translateWithMarian.sh$

```
1  #!/bin/bash
2
3  model=$1     # model.npz
4  vocabSrc=$2 # vocab.de.yml
5  vocabTgt=$3 # vocab.en.yml
6
7  marian-decoder -m $model -v $vocabSrc $vocabTgt \
8    --quiet-translation \
9    --beam-size 12 \
10   --normalize 0.6 \
11   --mini-batch 64 --maxi-batch-sort src --maxi-batch 100 \
12   --workspace 2500
```

W liniach od lini 7 wywołany zostaje program $marian-decoder$ należący do frameworka Marian przeprowadzający tłumaczenie W liniach 3-5 ścieżki przekazane w parametrach wywołania skryptu przypisane są do zmiennych wykorzystywanych dalej przy uruchamia-

niu programu pakietu Marian (w lini 7 skryptu 5.18). Parametr *quiettranslation* oznacza że podczas tłumaczenia nie będą wyświetlane żadne komunikaty (żaden log). Manipulacja pozostałymi parametrami służy poprawieniu wydajności i jakości tłumaczenia. Sam tekst do przetłumaczenia zostaje przekazany do programu ze skryptu nadrzędnego (z miejsca wywołania skryptu $translateWithMarian.sh$ w lini50. kodu źródłowego 5.17).

Rozdział 6

Ewaluacja modeli i wnioski

Projekt zakłada przeprowadzenie porównania modeli tłumaczy automatycznych. Płaszczyzną, na której może ono zachodzić, jest weryfikacja jakości tłumaczeń otrzymanych z wykorzystaniem testowanych translatorów. W niniejszym rozdziale zamieszczono zestawienie poprawności tych tłumaczeń oraz przedstawiono możliwą interpretację otrzymanych wyników.

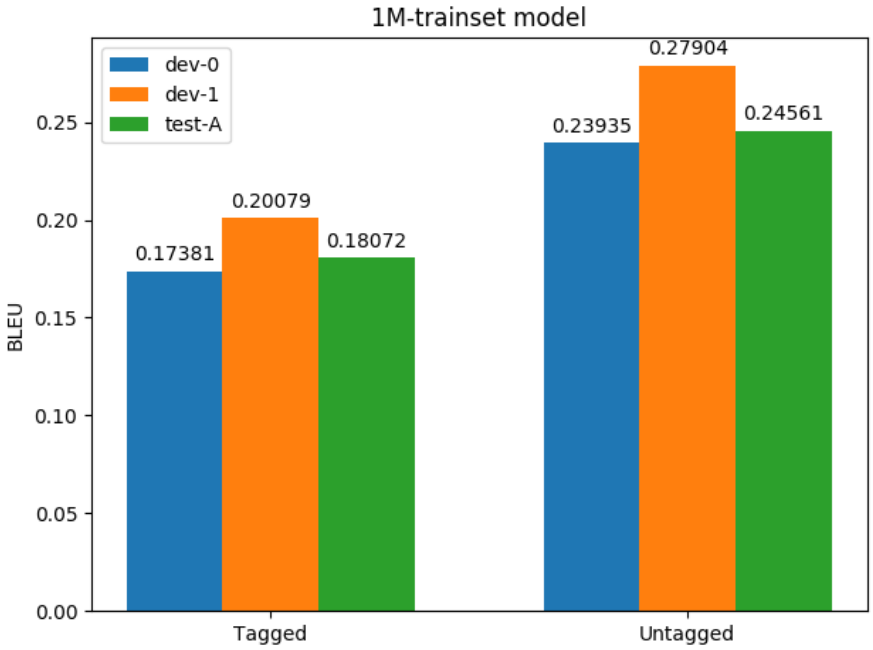
6.1. Pomiar jakości

Jakość modelu tłumacza automatycznego można weryfikować przez sprawdzenie, czy dostarczane przez niego tłumaczenia są dobre.

W przeprowadzonym badaniu za zbiory testowe posłużyły zbiory $dev-0$, $dev-1$ i $test-A$ pochodzące ze strony sympozjum EMNLP 2017 ([6]) a pobrane za pośrednictwem platformy Gonito.net (zbiory treningowe tłumaczy również pochodziły z tej lokalizacji).

Na przykładzie zgromadzonych zbiorów testowych zweryfikowano jakość tłumaczenia z wykorzystaniem czterech modeli tłumaczy: dwóch wytrenowanych z użyciem korpusów języka źródłowego i docelowego bez tagów (jeden model na korpusach o rozmiarze po 100 tysięcy, jeden na korpusach o rozmiarze po 1 milionie zdań/wierszy) oraz dwóch wytrenowanych z użyciem korpusów języka źródłowego i otagowanego języka docelowego (jeden model na korpusach o rozmiarze po 100 tysięcy, jeden na korpusach o rozmiarze po 1 milionie zdań/wierszy).

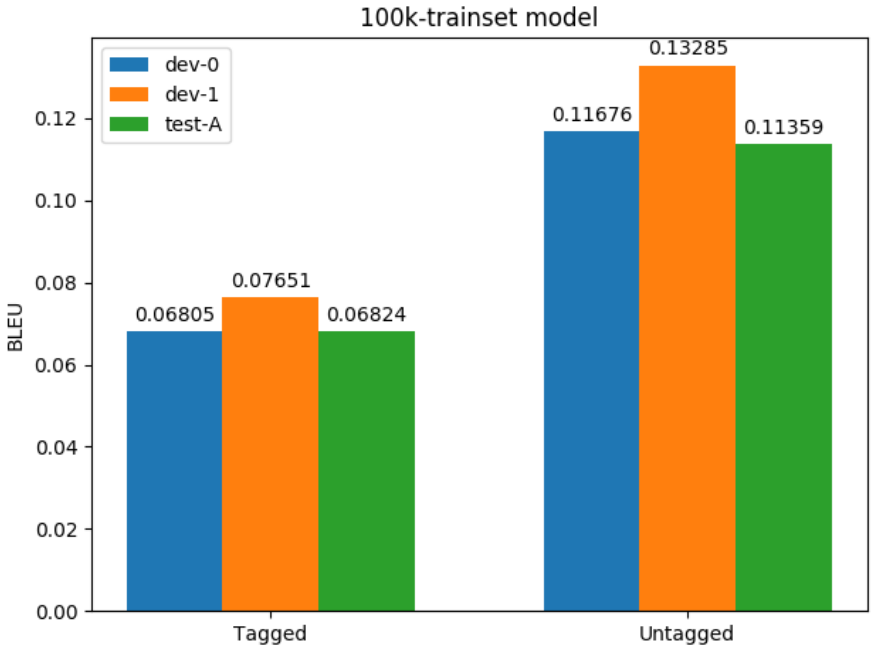
Do przeprowadzenia ewaluacji tłumaczeń wybrano jedną z podstawowych metryk służących ocenie jakości tłumaczenia — metrykę BLEU (opisano ją w rozdziale 1.2.2).



Rys. 6.1. Wartości BLEU — model 1mln zdań

Na wykresie zamieszczonym na rysunku 6.1 zebrano wyniki pomiarów przeprowadzonych dla trzech zbiorach testowych. Prawa strona wykresu (grupa *Untagged*) zawiera dane zebrane z badań przeprowadzonych na tłumaczeniach osiągniętych z modelu tłumacza wytrenowanego na standardowych zbiorach treningowych (zbiory nieotagowane). Pomimo pewnych rozbieżności prezentują one w miarę podobną jakość tłumaczenia (w okolicach 0.25.5 pkt. BLEU). Zaskakująco przedstawia się porównanie z lewą stroną wykresu (grupa oznaczona *Tagged*). Pomimo podobnych wzajemnych relacji między wartościami BLEU w obrębie grupy (dla wszystkich zbiorów treningowych ocena jakości to w przybliżeniu 18.5 pkt. BLEU) zauważalny jest generalny spadek oceny w stosunku do grupy *Untagged*. Sytuacja taka zachodzi zarówno w przypadku porównania ocen dla tych samych zbiorów (np. dla zbioru $dev-1$ po stronie *Untagged* BLEU wynosi ok. 0.28, a po stronie *Tagged* BLEU wynosi ok. 0.20), jak i przy porównywaniu oceny tłumaczeń różnych zbiorów (np. dla zbioru $dev-0$ po stronie *Untagged* BLEU wynosi ok. 0.24, a dla zbioru $dev-1$ po stronie *Tagged* BLEU wynosi ok. 0.20).

Przy drastycznym zmniejszeniu rozmiaru korpusów treningowych (na przykład przy zastosowaniu korpusów o rozmiarze po 100 tysięcy zadań/linii) poza naturalnym obniżeniem oceny tłumaczeń (w stosunku do rys. 6.1) relacje między grupami *Untagged* i *Tagged* pozostają podobne.



Rys. 6.2. Wartości BLEU — model 100tys zdań

6.2. Konkluzje

Przystępując do realizacji zadań niniejszej pracy postawiono roboczą hipotezę, że włączenie tagów do strony docelowej korpusu treningowego modelu tłumacza automatycznego może prowadzić do poprawienia parametrów tłumaczenia z jego wykorzystaniem. Prezentowana ocena tłumaczenia (rysunki 6.1 i 6.2), uzyskana metodą BLEU, kończąca prezentowaną w pracy procedurę sprawdzania wspomnianej hipotezy, zmusza jednak do przyjęcia raczej negatywnego stwierdzenia. Zdają się to potwierdzać zarówno ilościowe pomiary przedstawione na rysunkach 6.1 i 6.2, jak i jakościowa ocena wyników tłumaczenia. Tą drugą można sformułować choćby na podstawie przykładowych zdań z rysunku 6.3. Tłumaczenie uzyskane z użyciem modelu wytrenowanego z tagami w większości przypadków w dużo mniejszym stopniu przypomina tłumaczenie oczekiwane (*expected*). Pomimo zachowania pewnych podobieństw (również w obszarze znaczenia zdania) otrzymana translacja znacznie różniła się od wzorca. Na podstawie prób można było stwierdzić, że faktoryzowanie danych wyjściowych w neuronowym tłumaczeniu automatycznym, realizowane przez konstruowanie tłumacza z wykorzystaniem otagowanej strony docelowej korpusu treningowego, nie prowadzi do poprawy jakości tłumaczenia.

Na tym etapie badań trudno jednoznacznie wskazać przyczynę powyższej sytuacji. Dalsze postępowanie, które nie wchodziło w zakres niniejszej pracy, mogłoby obejmować:

expected:
1) Obama receives Netanyahu
2) The relationship between Obama and Netanyahu is not exactly friendly .
20)Students said they looked forward to his class .
24)At the same time , there were some inclinations of problems .

out (untagged):
1) Obama welcomes Netanyahu
2) The relationship between Obama and Netanyahu is not just friendly .
20)Students said that they enjoyed their teaching .
24)At the same time , there were some pointers to problematic behaviour

out (tagged):
1) Obama receives :
2) The relationship between Obama and the Commission is not just friendly .
20)Students said they have seen their lessons .
24)At the same time , there have been some problems about problematic behaviour .

Rys. 6.3. Tłumaczenia — przyklad z *dev*1

- faktoryzację strony źródłowej korpusu treningowego,
- modyfikację algorytmu trenowania modelu tłumacza przystosowaną do tagowania,
- połączenie tagów z poprzedzającymi słowami, aby podczas przetwarzania powstałe pary traktować jak jedno słowo,
- ewaluację z użyciem metryki innej niż BLEU, na przykład (GLEW, Meteor, WER).

Dodatek A

Kod źródłowy tokenizacji tekstów niemieckich

Kod źródłowy A.1. Tokenizacja zbiorów niemieckich

```
# -*- coding: utf-8 -*-

import sys
import time

print(sys.argv[0])

PACK_SIZE = 100000

# corpse_dir = '1M/tmp.de'
corpse_dir1 =
    '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.clean.de'
corpse_dir2 =
    '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.np.de'
corpse_dir3 = '/home/marek/master/gonito/wmt-2017/dev-0/in.np.tsv'
corpse_dir4 = '/home/marek/master/gonito/wmt-2017/dev-1/in.np.tsv'
corpse_dir5 = '/home/marek/master/gonito/wmt-2017/test-A/in.np.tsv'
corpse_dir = [corpse_dir1, corpse_dir2, corpse_dir3, corpse_dir4,
    corpse_dir5]

# tagged_dir = 'tmpTagged.de'
tagged_dir1 =
    '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.text.de'
```

```
20 tagged_dir2 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.text.de'
21 tagged_dir3 = '/home/marek/master/gonito/wmt-2017/dev-0/in.tg.de'
22 tagged_dir4 = '/home/marek/master/gonito/wmt-2017/dev-1/in.tg.de'
23 tagged_dir5 = '/home/marek/master/gonito/wmt-2017/test-A/in.tg.de'
24 tagged_dir = [tagged_dir1, tagged_dir2, tagged_dir3, tagged_dir4,
       tagged_dir5]
25
26
27 def start_test(n=None):
28     start = time.time()
29     import treetaggerpoll as ttp
30     import treetaggerwrapper as ttw
31
32     # p = ttp.TaggerProcessPoll(workerscount=n, TAGLANG="de")
33
34     currLine = 0    # line being processed (within PACK_SIZE)
35     l = 0    # total number of sentences tagged
36
37     for j, cd in enumerate(corpse_dir):
38         p = ttp.TaggerProcessPoll(workerscount=n, TAGLANG="de")
39         lap = time.time()
40         # in_file = open(cd, 'r')
41         with open(cd, 'r') as in_file:
42             print('\nRare german corpse: '+cd+' opened to read')
43
44             taggSet = []
45             print('Tokenize sentences - multiprocessing...')
46             for sentence in in_file.readlines():
47                 # print('sen:'+sentence, end='')
48                 taggSet.append(p.tag_text_async(sentence,
                       prepronly=True, notagurl=True,
49                                                 notagemail=True,
                                                       notagip=True,
                                                       notagdns=True,
                                                       nosgmlsplit=True))
50         print('done in  {:0.2f} seconds\n'.format((time.time() -
               lap)))
51
```

```
52          # out_file = open(tagged_dir[j], 'w')
53          with open(tagged_dir[j], 'w') as out_file:
54              print(tagged_dir[j]+' opened to write')
55              print('Make tokens & write file...')
56              for i, r in enumerate(taggSet):
57                  r.wait_finished()
58                  # print(i, str(r.result))
59                  currLine = currLine+1
60
61                  for token in r.result:
62                      # print(i, token+' ', end='')
63                      out_file.write(token+' ')
64                  # print()
65                  out_file.write('\n')
66
67                  if (currLine == PACK_SIZE):
68                      currLine = 0
69                      print('line tokenized:', i+1, '\tfirst word:',
                            r.result[0])
70
71                  taggSet[i] = None
72              print('Tokenized', i+1, 'lines')
73          mid_tim = time.gmtime(time.time() - lap)
74          print(
75              'done in  {:d}h {:d}m {:d}sec
                    -----'.format(mid_tim[3], mid_tim[4], mid_tim[5]))
76          p.stop_poll()
77      print('\n----------END----------')
78
79      tim = time.gmtime(time.time() - start)
80      print('processed in  {:d}h {:d}m {:d}sec\n'.format(tim[3],
            tim[4], tim[5]))
81      # print('in  {:0.2f} seconds\n'.format((time.time() - start)))
82
83
84  if __name__ == '__main__':
85      print('---------START---------')
86
87      if len(sys.argv) >= 2:
```

```
88          nproc = int(sys.argv[1])
89          print('nproc=', nproc)
90      else:
91          nproc = None
92          print('nproc=', nproc, '(CPUs number)')
93
94      start_test(nproc)
95      # print('\n----------END----------')
```

Dodatek B

Kod źródłowy tagowania tekstów angielskich

Kod źródłowy B.1. Tagowanie i tokenizacja zbiorów angielskich

```
1  # -*- coding: utf-8 -*-
2
3  import sys
4  import time
5
6  print(sys.argv[0])
7
8  PACK_SIZE = 100000
9
10 # corpse_dir = '1M/tmp.en'
11 corpse_dir1 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.clean.en'
12 corpse_dir2 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.np.en'
13 corpse_dir = [corpse_dir1, corpse_dir2]
14
15 # tagged_dir = 'tmpTagged.en'
16 tagged_dir1 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/corpus.tg.en'
17 tagged_dir2 =
       '/home/marek/master/gonito/wmt-2017/corpse/1M/validSet.tg.en'
18 tagged_dir = [tagged_dir1, tagged_dir2]
19
```

```
20
21 def start_test(n=None):
22     start = time.time()
23     import treetaggerpoll as ttp
24     import treetaggerwrapper as ttw
25
26     # p = ttp.TaggerProcessPoll(workerscount=n, TAGLANG="en")
27
28     currLine = 0    # line being processed (within PACK_SIZE)
29     l = 0    # total number of sentences tagged
30
31     for j, cd in enumerate(corpse_dir):
32         p = ttp.TaggerProcessPoll(workerscount=n, TAGLANG="en")
33         lap = time.time()
34         # in_file = open(cd, 'r')
35         with open(cd, 'r') as in_file:
36             print('\nRare english corpse: '+cd+' opened to read')
37
38             taggSet = []
39             print('Tag sentences - multiprocessing...')
40             for sentence in in_file.readlines():
41                 # print('sen:'+sentence, end='')
42                 taggSet.append(p.tag_text_async(sentence,
                        notagurl=True,
43                                                 notagemail=True,
                                                    notagip=True,
                                                    notagdns=True,
                                                    nosgmlsplit=True))
44         print('done in  {:0.2f} seconds\n'.format((time.time() -
               lap)))
45
46         # out_file = open(tagged_dir[j], 'w')
47         with open(tagged_dir[j], 'w') as out_file:
48             print(tagged_dir[j]+' opened to write')
49             print('Make tags & write file...')
50             for i, r in enumerate(taggSet):
51                 r.wait_finished()
52                 # print(i, str(r.result))
53                 currLine = currLine+1
```

```
54                  taggSen = ttw.make_tags(
55                      r.result, exclude_nottags=True,
                            allow_extra=True)
56                  for token in taggSen:
57                      # print(token[0]+' '+token[1]+' ', end='')
58                      out_file.write(token[0]+' '+token[1]+' ')
59                  # print()
60                  out_file.write('\n')
61
62                  if (currLine == PACK_SIZE):
63                      currLine = 0
64                      print('line tagged:', i+1, '\tfirst word:',
65                            taggSen[0][0], '\ttag:', taggSen[0][1])
66
67                  taggSet[i] = None
68              print('Tagged', i+1, 'lines')
69          mid_tim = time.gmtime(time.time() - lap)
70          print(
71              'done in  {:d}h {:d}m {:d}sec
                    -----'.format(mid_tim[3], mid_tim[4], mid_tim[5]))
72          p.stop_poll()
73      print('\n----------END----------')
74
75      tim = time.gmtime(time.time() - start)
76      print('processed in  {:d}h {:d}m {:d}sec\n'.format(tim[3],
            tim[4], tim[5]))
77      # print('in  {:0.2f} seconds\n'.format((time.time() - start)))
78
79
80  if __name__ == '__main__':
81      print('---------START---------')
82
83      if len(sys.argv) >= 2:
84          nproc = int(sys.argv[1])
85          print('nproc=', nproc)
86      else:
87          nproc = None
88          print('nproc=', nproc, '(CPUs number)')
89
```

```
90      start_test(nproc)
91      # print('\n----------END----------')
```

Bibliografia

[1] Kim jest deepl?, https://www.deepl.com/pro-faq.html. [Dostęp: 7 marca 2019].

[2] Moses, http://www.statmt.org/moses/. [Dostęp: 12 października 2018].

[3] Penn treebank tagset, https://www.sketchengine.eu/penn-treebank-tagset/. [Dostęp: 5 września 2018].

[4] Pos tags, https://www.sketchengine.eu/pos-tags/. [Dostęp: 10 grudnia 2018].

[5] W porównaniu z konkurencją, https://www.deepl.com/press.html. [Dostęp: 8 marca 2019].

[6] EMNLP 2017 SECOND CONFERENCE ON MACHINE TRANSLATION, Copenhagen, Denmark, September 7-8 2017. http://www.statmt.org/wmt17/translation-task.html [Dostęp: 15 lipca 2018].

[7] Marian NMT Documentation – marian-decoder, https://marian-nmt.github.io/docs/cmd/marian-decoder/, December 2018. [29 grudnia 2018].

[8] Marian NMT Documentation, https://marian-nmt.github.io/docs/, August 2019. [Dostęp: 28 sierpnia 2019].

[9] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate, May 2016. https://arxiv.org/pdf/1409.0473.pdf [Dostęp: 17 listopada 2017].

[10] Boschetti, A., and Massaron, L. *Python Podstawy Nauki o Danych, wydanie II.* Helion, Gliwice, 2017, p. 223.

[11] Gage, P. A new algorithm for data com-pression. *C Users J. 12*, 2 (1994), 23–38.

[12] Goldberg, Y. *Neural Network Methods for Natural Language Processing.* Morgan & Claypool, 2017.

[13] Grajek, M. *Enigma, Bliżej Prawdy.* REBIS, Poznań, 2007.

[14] Junczys-Dowmunt, M., Grundkiewicz, R., Dwojak, T., Hoang, H., Heafield, K., Neckermann, T., Seide, F., Germann, U., Fikri Aji, A., Bogoychev, N., Martins, A. F. T., and Birch, A. Marian: Fast neural machine translation in C++. In *Proceedings of ACL 2018, System Demonstrations* (Melbourne, Australia, July 2018), Association for Computational Linguistics, pp. 116–121.

[15] Koehn, P. *Statistical Machine Translation.* Cambridge University Press, Cambridge, UK, 2013.

[16] Koehn, P. *Statistical Machine Translation.* Cambridge University Press, Cambridge, UK, 2013, ch. 1.2.1.

[17] Koehn, P. *Statistical Machine Translation. Draft of Chapter 13: Neural Machine Translation.* 2017, ch. 13.5.1 Encoder-Decoder Approach. https://arxiv.org/pdf/1709.07809.pdf [Dostęp: 15 marca 2018].

[18] Nadejde, M., Reddy, S., Sennrich, R., Dwojak, T., Junczys-Dowmunt, M., Koehn, P., and Birch, A. Predicting target language ccg supertags improves neural machine translation in proceedings of the conference on machine translation (wmt) research papers. second conference on machine translation (wmt), 2017. Association for Computational Linguistics, Copenhagen, Denmark https://arxiv.org/pdf/1702.01147.pdf [Dostęp: 18 stycznia 2018.

[19] Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Cloth-iaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. Dynet: The dynamic neural network toolkit. https://arxiv.org/pdf/1701.03980.pdf [Dostęp: 7 stycznia 2018].

[20] Och, F. Statistical machine translation live, April 2006. https://ai.googleblog.com/2006/04/statistical-machine-translation-live.html [Dostęp: 4 sierpnia 2019].

[21] Pointal, L. *TreeTagger Python Wrapper's documentation!* CNRS - LIMSI. https://treetaggerwrapper.readthedocs.io/en/latest/ [Dostęp: 8 stycznia 2018].

[22] Santorini, B. Part-of-speech tagging guidelines for the penn treebank project, 1991. https://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/data/Penn-Treebank-Tagset.pdf July 1990 MS-CIS-90-47 LINC LAB 178 [Dostęp: 20 lutego 2018].

[23] Schmid, H. *TreeTagger - a part-of-speech tagger for many languages.* University of Stuttgart, Stuttgart, Germany. https://www.cis.uni-muenchen.de/ schmid/tools/TreeTagger/ [Dostęp: 16 listopada].

[24] Sennrich, R., Firat, O., Cho, K., Birch, A., Haddow, B., Hitshler, J., Junczys-Dowmunt, M., Laubli, S., Barone, A. V. M., Mokry, J., and Nadeje, M. Nematus: a toolkit for neural machine translation, 2017. https://arxiv.org/pdf/1703.04357.pdf [Dostęp: 10 stycznia 2018].

[25] Sennrich, R., and Haddow, B. Linguistic input features improve neural machine translation, 2016. https://arxiv.org/pdf/1606.02892.pdf [Dostęp 3 lutego 2018].

[26] Sennrich, R., Haddow, B., and Birch, A. *Neural Machine Translation of Rare Words with Subword Units.* 2016, ch. 3.2. https://arxiv.org/pdf/1508.07909.pdf [Dostęp: 9 listopada 2018].

[27] Turovsky, B. Found in translation: More accurate, fluent sentences in google translate, https://www.blog.google/products/translate/found-translation-more-accurate-fluent-sentences-google-translate/, November 2016. [Dostęp: 4 sierpnia 2019].

Spis rysunków

1.1. Oceny tłumaczeń dokonane przez ludzi – wartości procentowe oznaczają dla jakiej części oceniających dany translator został uznany za najlepszy . 13
1.2. Ocena BLEU 13
1.3. Precision — przykład 14
1.4. BLEU — przykład 15

2.1. Fragment korpusu (niemiecki) 21
2.2. Fragment korpusu (angielski) 21
2.3. Fragment zbioru walidacyjnego (niemiecki) 22
2.4. Fragment zbioru walidacyjnego (angielski) 23
2.5. Tekst z tagami - przykład 24

3.1. Tokenizacja 33
3.2. Naiwna tokenizacja 34
3.3. Słowa niemieckie 35

4.1. Tekst z tagami - przykład 46
4.2. Kilka znaczeń wyrazów 46

6.1. Wartości BLEU — model 1mln zdań 79
6.2. Wartości BLEU — model 100tys zdań 80
6.3. Tłumaczenia — przyklad z *dev*1 81

Spis tabel

1.1. Ewaluacja tłumaczeń 14
1.2. Ewaluacja tłumaczeń 16

2.1. wszystkie parametry metody *TaggerProcessPoll* 25
2.2. Wszystkie parametry metody *tag_text_async* 26

3.1. parametry programu clean-corpus-n.perl 32
3.2. Parametry metody *tag_text_async* dla tokenizacji zbiorów 37

4.1. Zestaw tagów Penn Treebank [3] 49

5.1. Parametry metody *tag_text_async* dla tagowania zbiorów 63