

# Realizacja prostego silnika reguł walidacyjnych przy użyciu technik NLP

Mariusz Wójcik

16 czerwca 2019

## Część I

# Przygotowanie i trening modelu

# 1 Początki

Jakiś czas temu miałem okazję obejrzeć film, który zrobił na mnie ogromne wrażenie. „*Arrival - Nowy Początek*” w reżyserii Denisa Villeneuve’a - to obraz niezwykle. Porusza on problematykę szerokopojętej, wielopłaszczyznowej komunikacji (a czasem skutków jej braku) . W niesamowicie sugestywny i obrazowy sposób pokazuje mechanizm kształtowania się podstaw wspólnego języka i nawiązywania kontaktu. Proces stopniowego budowania u wspólnionych modeli pojęciowych prowadzący do porozumiewania się tym samym językiem wydał mi się tak logiczny i uporządkowany, że sprawiał wrażenie niemal algorytmicznego... Pamiętam swoją myśl, że skoro możliwe jest tak precyzyjne określenie reguł stojących u podstaw nawiązania skutecznej komunikacji, to droga do zbudowania inteligentnych maszyn porozumiewających się z nami „ludzkim” językiem wydaje się już bardzo krótka.

Do niedawna wydawało mi się, że porozumiewanie się językiem naturalnym jest domeną przynależną wyłącznie człowiekowi. Miałem poczucie, że prace nad komputerowym przetwarzaniem języka naturalnego mają wymiar wyłącznie akademicki. Okazało się jednak, że dynamiczny rozwój algorytmów sztucznej inteligencji i przetwarzania maszynowego dotknął również tej dziedziny. Gdzieś na styku matematyki, informatyki i lingwistyki wykształciła się dziedzina, która funkcjonuje jako *NLP* (*ang. natural language processing* ).

Techniki NLP koncentrują się na analizie, przekształcaniu i generowaniu języka naturalnego. Dzięki nim komputery nabywają umiejętności nie tylko analizy tekstu, ale również nauki i wyciągania wniosków. Dają one możliwość analizy nie tylko składni zdań, ale również doszukiwania się ich znaczeń i ukrytych pomiędzy słowami intencji. Czasami uświadamiam sobie że to wszystko razem brzmi jak czysta fantastyka. Bo jak niby sens, znaczenie i intencje można przeliczyć na liczby i twardo zakotwiczyć w dziedzinie algebry liniowej ?

Tajemnicy tej uchyla jedna z najciekawszych książek, jaką miałem przyjemność ostatnio czytać, mianowicie „*Natural Language Processing in Action*”. Jest to bardzo przystępnie napisany przewodnik, dzięki któremu łatwiej oswoić się z podstawowymi prawami rządzącymi światem NLP. Pozycja nie traktuje o rzeczach najłatwiejszych, a mimo to czyta się ją z dużą przyjemnością.

Z teorią często jest tak, że w którymś momencie chciałoby się ją zobaczyć w praktyce. Z tej potrzeby zrodził się pomysł na aplikację, którą możnaby zrealizować przy użyciu technik i algorytmów NLP. Przyszedł mi do głowy generator kodu aplikacji, który byłby w stanie przekształcić tekst napisany językiem zbliżonym do naturalnego bezpośrednio do kodu wykonywalnego. Oczywiście zakładałam że tego typu rozwiązanie miałoby zastosowanie do jakiegoś ściśle określonego aspektu działającej aplikacji, np. walidacji dokumentu, czy sprawdzania reguł poprawności modelu dziedziny. I tak właśnie powstał mój miniprojekt, którego celem jest zobaczenie o co tak naprawdę chodzi z tym NLP . :) . Zapraszam do zapoznania się z założeniami i otrzymanymi wynikami.

Mam świadomość, że jeśli chodzi o NLP, jestem na początku drogi. Nie mogę powiedzieć nawet tego, że udało mi się zrobić jeden krok, ale wiem jedno... podróż zapowiada się naprawdę imponująco...

## 2 Realizacja

Do realizacji moich założeń wybrałam napisaną w Javie bibliotekę *Apache OpenNLP*. Dostarcza ona narzędzi realizujących wiele aspektów przetwarzania języka naturalnego. W moim projekcie skupię się technice nazywanej *Named Entity Recognition (NER)*.

Polega ona na rozpoznawaniu w tekście określonych bytów nazwanych. Najczęściej są to imiona, nazwiska, nazwy własne itp. W moim przypadku chciałbym stworzyć własny model, który zostanie przyuczony do rozpoznawania poszczególnych elementów konstrukcji reguły walidacyjnej (takich jak słowa kluczowe rozpoczynające i kończące bloki, operatory, akcje i ich parametry).

Żeby to osiągnąć konieczne jest przygotowanie odpowiednio opisanej próbki uczącej, a następnie wykorzystanie jej do treningu modelu.

### 3 Abstrakcyjny model reguły

Przygotowanie próbki rozpocznę od opracowania schematu reguły.

Na początek wypiszę sobie kilka przykładowych reguł walidacyjnych.

1. *Jeśli wiek\_pacjenta jest większy od 18 wtedy zgłoś błąd „Pacjent jest osobą dorosłą.”, w przeciwnym wypadku wyświetl komunikat „Pacjent został zakwalifikowany do leczenia pediatrycznego.”.*
2. *Jeśli data\_kwalifikacji jest mniejsza od '01-01-2019' wtedy zgłoś wyjątek „Data sprzed roku 2019.”, w przeciwnym wypadku sprawdź regułę RS-001.*
3. *Gdy saldo\_rachunku jest większe od 100 oraz saldo\_rachunku jest mniejsze niż 1000 wtedy wyświetl komunikat „Saldo rachunku jest prawidłowe.”, w przeciwnym razie zgłoś błąd „Nieprawidłowe saldo rachunku”.*
4. *Jeśli data\_teraz jest niewiększa niż data\_ważności wyświetl komunikat „Wniosek jest aktualny.” w przeciwnym wypadku zgłaszaj błąd „Wniosek utracił ważność”.*

Przyjmuję uproszczenie, że każda rozpoznawana reguła składała się będzie z trzech wyróżnialnych bloków:

$\underbrace{\text{WARUNKI}}_{\text{Warunki}} \underbrace{\text{AKCJA\_TAK}}_{\text{Akcja Tak}} \underbrace{(\text{AKCJA\_NIE})}_{\text{Akcja Nie}}?$

Poszczególne bloki oddzielone będą od siebie słowami kluczowymi oznaczającymi rozpoczęcie i zakończenie bloku.

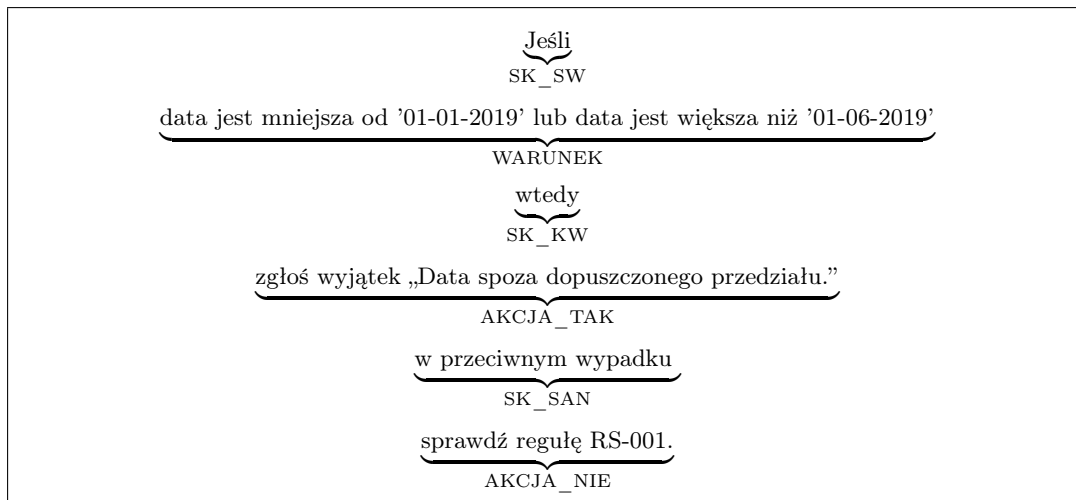
W celu ich wyróżnienia wprowadzam następujące oznaczenia:

1. SK\_SW - Start sekcji warunku
2. SK\_KW - Koniec sekcji warunku
3. SK\_SAN - Start sekcji akcji wykonywanej przy niespełnionym warunku

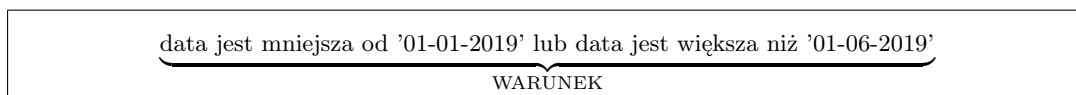
Schemat reguły przyjmuje następującą postać:

$\underbrace{\text{SK\_SW}}_{\text{SK\_SW}} \underbrace{\text{WARUNKI}}_{\text{WARUNKI}} \underbrace{\text{SK\_KW}}_{\text{SK\_KW}} \underbrace{\text{AKCJA\_TAK}}_{\text{AKCJA\_TAK}} \underbrace{(\text{SK\_SAN AKCJA\_NIE})}_{\text{SK\_SAN AKCJA\_NIE}}?$

Rzut oka na przykład:



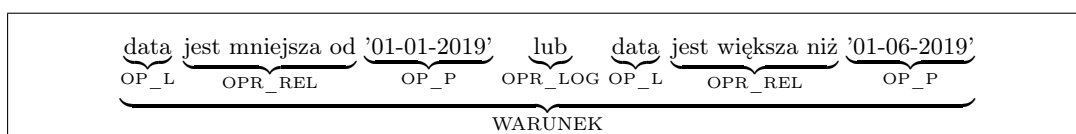
Ponieważ kluczowe jest właściwe rozpoznanie sekcji warunku, chciałbym wyłączyć go przed nawias i przez chwilę skupić się wyłącznie na nim.



Na początek trzeba zauważyć, że powyższa sekcja składa się z dwóch niezależnych wyrażeń warunkowych połączonych operatorem logicznym *lub*. Każdy z pojedynczych warunków składa się z kolei z operatora relacyjnego (*jest mniejsza*, *jest większa*), oraz z dwóch operandów (*lewego i prawego*). Wprowadzam więc następujące oznaczenia:

1. OP\_L - Operand lewy
2. OPR\_REL - Operator relacyjny
3. OP\_P - Operand prawy
4. OPR\_LOG - Operator logiczny

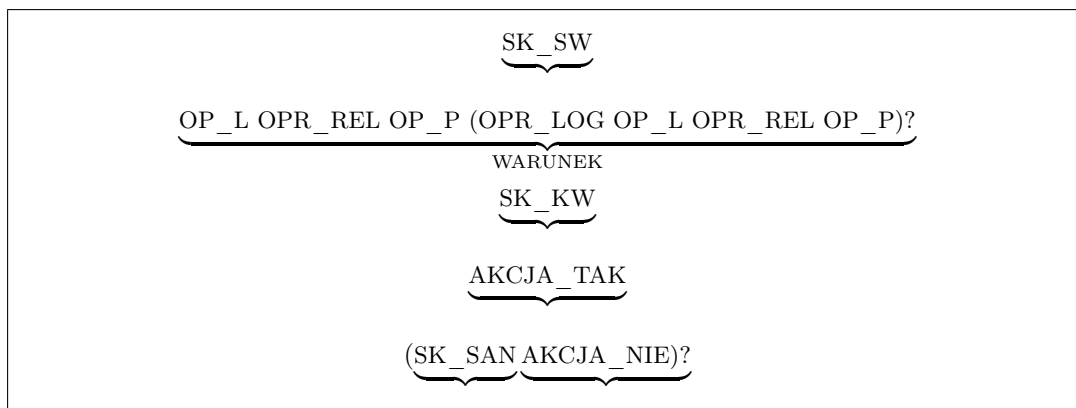
Po podstawieniu, przykładowy warunek można



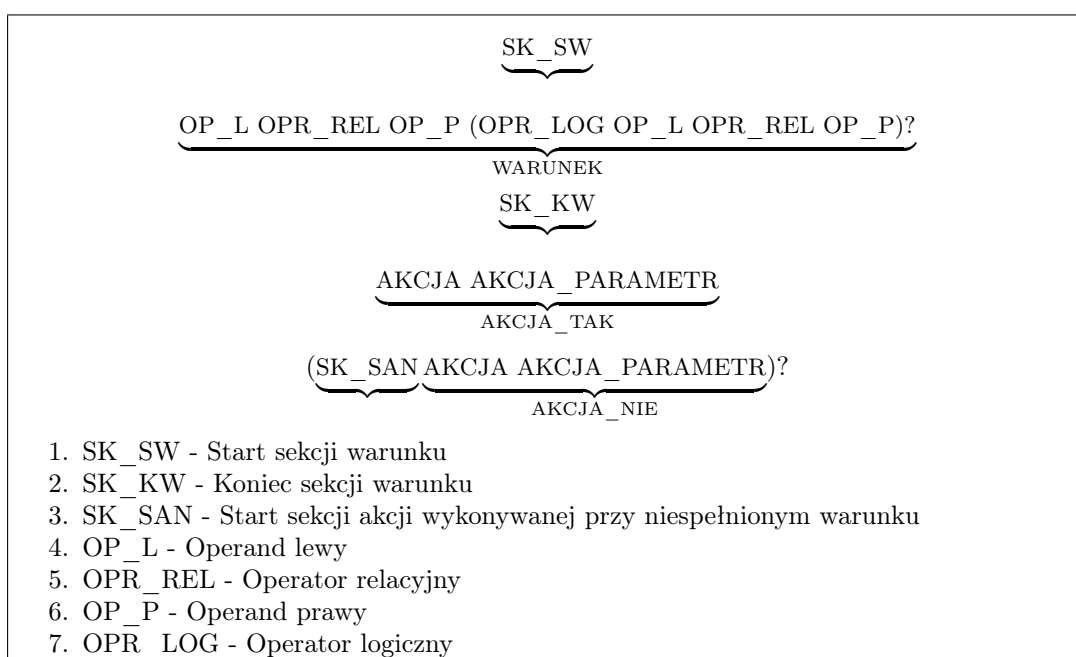
Zatem zapis symboliczny sekcji warunkowej będzie wyglądał następująco:



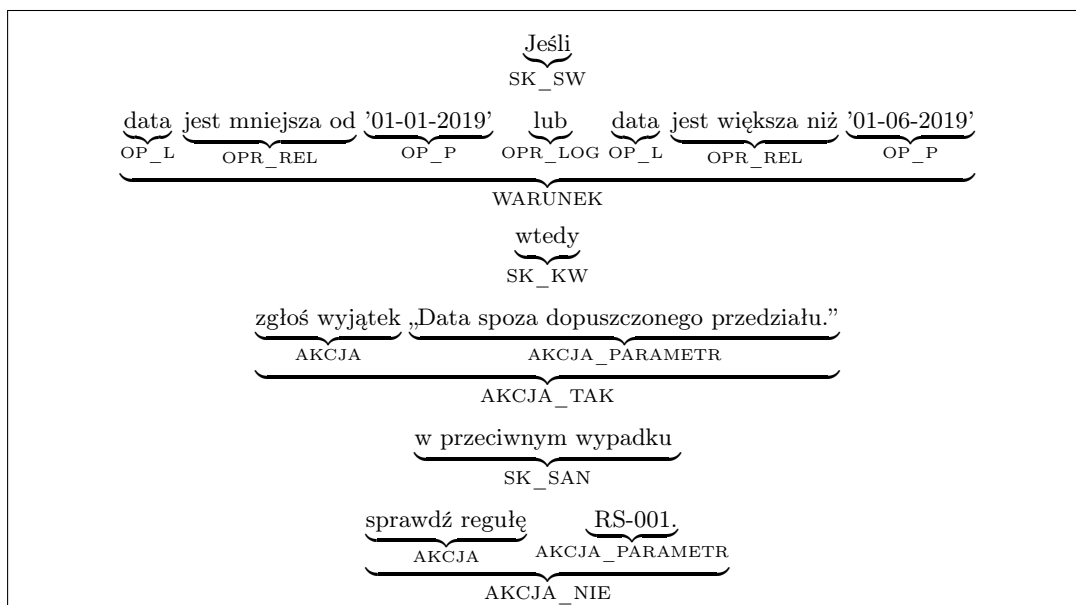
Po dokonaniu podstawienia w sekcji *WARUNKI* abstrakcyjny model reguły przyjmie następującą postać:



Jeśli chodzi o akcje, to sprawa wydaje się prostsza, bo każda z nich składa się z części mówiącej o tym co ma być zrobione (AKCJA), oraz z jakim parametrem ma być wykonane (AKCJA\_PARAMETR). Ostatecznie więc, po wykonaniu podstawienia nasz model przyjmie następującą, ostateczną formę:



I jeszcze spojrzenie na przykład:



## 4 Przygotowanie próbki uczącej

Mechanizmy NER wchodzące w skład *OpenNLP* pozwalają na stworzenie własnego modelu i przyuczenie go do rozpoznawania specyficznych bytów domenowych. Próbka ucząca jest dosyć obszernym zbiorem przykładów (dokumentacja OpenNLP mówi o minimum 15 tyś. zdań), w których w specjalny sposób otagowane zostały kluczowe frazy.

```
<START:SK_SW> jeśli <END> <START:OP_L> xxx <END> <START:OPR_REL> jest
większy niż <END> <START:OP_P> xxx <END> <START:SK_KW> wtedy <END>
<START:AKCJA> zgłoś błąd <END> <START:AKCJA_PARAMETR> xxx <END> .
```

Byty, które docelowo mają być rozpoznawane przez model należy umieścić pomiędzy tagami *<START:NAZWA\_BYTU>* i *<END>*. Dodatkowo każda reguła danych uczących zbudowana jest według schematu omawianego wcześniej abstrakcyjnego modelu reguły. Zgodne z nim są również nazwy encji. W przypadku tych części reguły, które są zmienne i specyficzne dla każdej instancji (takie jak komentarze, nazwy operandów, wszelkie parametry) użyłem frazy *xxx*, która oznacza że będzie tu coś, o czym na tym etapie nie możemy nic powiedzieć (znamy tylko pozycję tego tokena względem innych encji).

Wygenerowanie próbki uczącej okazało się zagadnieniem samym w sobie. Do tego celu napisałem aplikację pythonową, która w danych wejściowych otrzymuje przewidywane przykładowe wartości encji, a następnie otagowuje je, tworzy ich iloczyny kartezjańskie i ostatecznie konstruuje z nich zdanie zgodne z założonym schematem.

Poniżej znajdują się wartości poszczególnych encji użyte to wygenerowania danych uczących.

$$SK\_SW = \begin{cases} \text{jeśli} \\ \text{gdy} \\ \text{jeżeli} \end{cases}$$

$$OPR\_REL = \{ \text{nie} \} * \begin{cases} \text{jest równy} \\ \text{jest równa} \\ \text{jest równe} \\ \text{jest większy} \\ \text{jest mniejszy} \\ \text{jest różny} \\ \text{jest większa} \\ \text{jest mniejsza} \\ \text{jest różna} \\ \text{jest większe} \\ \text{jest mniejsze} \\ \text{jest różne} \end{cases} \begin{cases} \text{niż} \\ \text{od} \end{cases}$$

$$SK\_KW = \begin{cases} \text{wtedy} \\ \text{to} \end{cases}$$

$$SK\_SAN = \{ \text{w przeciwnym wypadku} \}$$

$$AKCJA = \left\{ \begin{array}{l} \text{zgłoś błąd} \\ \text{zgłoś błąd walidacji} \\ \text{raportuj błąd} \\ \text{wyświetl komunikat} \\ \text{sprawdź regułę} \\ \text{sprawdzaj regułę} \end{array} \right\}$$

Przykładowe, bardziej złożone reguły utworzone opisaną wcześniej techniką:

1.  $\langle START:SK\_SW \rangle$  jeśli  $\langle END \rangle \langle START:OP\_L \rangle xxx \langle END \rangle \langle START:OPR\_REL \rangle$  jest większy niż  $\langle END \rangle \langle START:OP\_P \rangle xxx \langle END \rangle \langle START:SK\_KW \rangle$  wtedy  $\langle END \rangle \langle START:AKCJA \rangle$  zgłoś błąd  $\langle END \rangle \langle START:AKCJA\_PARAMETR \rangle xxx \langle END \rangle \langle START:SK\_SAN \rangle$  w przeciwnym wypadku  $\langle END \rangle \langle START:AKCJA \rangle$  sprawdzaj regułę  $\langle END \rangle \langle START:AKCJA\_PARAMETR \rangle xxx \langle END \rangle$ .
2.  $\langle START:SK\_SW \rangle$  jeśli  $\langle END \rangle \langle START:OP\_L \rangle xxx \langle END \rangle \langle START:OPR\_REL \rangle$  nie jest mniejsza od  $\langle END \rangle \langle START:OP\_P \rangle xxx \langle END \rangle \langle START:OPR\_LOG \rangle$  oraz  $\langle END \rangle \langle START:OP\_L \rangle xxx \langle END \rangle \langle START:OPR\_REL \rangle$  nie jest równy  $\langle END \rangle \langle START:OP\_P \rangle xxx \langle END \rangle \langle START:SK\_KW \rangle$  wtedy  $\langle END \rangle \langle START:AKCJA \rangle$  zgłoś błąd  $\langle END \rangle \langle START:AKCJA\_PARAMETR \rangle xxx \langle END \rangle \langle START:SK\_SAN \rangle$  w przeciwnym wypadku  $\langle END \rangle \langle START:AKCJA \rangle$  zgłoś błąd  $\langle END \rangle \langle START:AKCJA\_PARAMETR \rangle xxx \langle END \rangle$ .
3.  $\langle START:SK\_SW \rangle$  jeśli  $\langle END \rangle \langle START:OP\_L \rangle xxx \langle END \rangle \langle START:OPR\_REL \rangle$  jest większy niż  $\langle END \rangle \langle START:OP\_P \rangle xxx \langle END \rangle \langle START:OPR\_LOG \rangle$  lub  $\langle END \rangle \langle START:OP\_L \rangle xxx \langle END \rangle \langle START:OPR\_REL \rangle$  jest różny od  $\langle END \rangle \langle START:OP\_P \rangle xxx \langle END \rangle \langle START:SK\_KW \rangle$  wtedy  $\langle END \rangle \langle START:AKCJA \rangle$  zgłoś błąd  $\langle END \rangle \langle START:AKCJA\_PARAMETR \rangle xxx \langle END \rangle \langle START:SK\_SAN \rangle$  w przeciwnym wypadku  $\langle END \rangle \langle START:AKCJA \rangle$  zgłoś błąd walidacji  $\langle END \rangle \langle START:AKCJA\_PARAMETR \rangle xxx \langle END \rangle$ .

Liczność próbki użytej do treningu modelu ustawiłem na poziomie ok 20 000 rekordów.

## 5 Trening modelu

Proces trenowania modelu realizowany jest przez niewielką aplikację (*Kotlin, OpenNLP NER API*), której zadaniem jest dostarczenie danych próbki, ustawienie parametrów uczenia, wystartowanie procesu trenowania, oraz zapisanie wygenerowanego binarnego pliku modelu. Najistotniejszy fragment aplikacji przedstawiony jest na listingu.

```
private fun trenujModel(aZbiorRegul:Path): TokenNameFinderModel {
    // reading training data
    var inputFactory: InputSteamFactory?
    try {
        inputFactory =
            MarkableFileInputSteamFactory(aZbiorRegul.toFile())
    } catch (e: FileNotFoundException) {
        throw IllegalArgumentException(e)
    }

    var sampleStream: ObjectStream<*>?

    sampleStream = NameSampleDataStream(
        PlainTextByLineStream(inputFactory, StandardCharsets.UTF_8))

    // setting the parameters for training
    val params = TrainingParameters()
    params.put(TrainingParameters.ITERATIONS_PARAM, 70)
    params.put(TrainingParameters.CUTOFF_PARAM, 1)

    // training the model using TokenNameFinderModel class
    var nameFinderModel: TokenNameFinderModel?
    try {
```



```

        nameFinderModel = NameFinderME.train("en"
        , null
        , sampleStream
        , params
        , TokenNameFinderFactory.create(null
        , null
        , mutableMapOf<String, Any>()
        , BioCodec()))

    return nameFinderModel

} catch (e: IOException) {
    throw IllegalArgumentException(e)
}
}

```

Raport podsumowujący pojedynczą sesję treningową wykonaną w oparciu o przygotowaną wcześniej próbkę wygląda następująco:

```

> Task :app-modul-silnik-regul:wytrenujModel
=====encje_reguly_probka_uczaca.reg
Indexing events with TwoPass using cutoff of 1

Computing event counts... done. 259140 events
Indexing... done.
Collecting events... Done indexing in 6,59 s.
Incorporating indexed data for training...
done.
Number of Event Tokens: 259140
Number of Outcomes: 13
Number of Predicates: 654
Computing model parameters...
Performing 70 iterations.
1: . (259100/259140) 0.9998456432816238
2: . (259140/259140) 1.0
3: . (259140/259140) 1.0
4: . (259140/259140) 1.0
5: . (259140/259140) 1.0
Stopping: change in training set accuracy less than 1.0E-5
Stats: (259140/259140) 1.0
...done.
Compressed 654 parameters to 322
79 outcome patterns
Trained model saved to file in location=>src/main/resources/modelnlp/encje_reguly_model.bin

```

Trening modelu jest krokiem kończącym etap przygotowawczy. Zapisany plik binarny jest gotowy do użycia go w aplikacji. W następnej części postaram się opisać jak można posłużyć się nim do rozwiązania konkretnego problemu.

# Część II

## Aplikacja

## 6 Architektura aplikacji

Aplikacja składa się z dwóch warstw-części GUI i części serwerowej. Część serwerowa odpowiedzialna jest za całościową analizę reguł, oraz generowanie kodu. Jest ona napisana w technologii Spring Boot, w języku Kotlin. Warstwa GUI stworzona została w Angularze. Komunikacja odbywa się przy pomocy serwisów REST. Z punktu widzenia użytkownika aplikacja składa się z trzech wyraźnie wyodrębnionych części

1. Sekcji danych wejściowych
2. Sekcji wyników analizy NLP
3. Sekcji wygenerowanego kodu

### 6.1 Sekcja danych wejściowych

W tej części aplikacji użytkownik może zarządzać danymi, jakie poddawane są analizie NLP. Interfejs aplikacji umożliwia dodawanie nowych reguł, modyfikację treści istniejących, oraz kasowanie reguł.



Rysunek 1: Główne okno aplikacji - sekcja definicji reguł wejściowych

## 6.2 Sekcja prezentacji wyników analizy NLP

Ten ekran jest nieco ciekawszy od poprzedniego. Prezentuje wyniki analizy reguł wykonanej w oparciu o algorytmy uczenia maszynowego dostępne w ramach biblioteki *OpenNLP*. Wyniki analizy prezentowane są w tabelach według następującego schematu:

1. **Reguła wejściowa** - informacja o treści reguły poddawanej analizie algorytmicznej.
2. **Wyodrębnione komunikaty** - jest to lista komunikatów użytkownika, które zostały odnalezione w regule. W celu uproszczenia przetwarzania sekwencji przez algorytmy NLP komunikaty są wyodrębniane i zastępowane słowem - markerem identyfikującym komunikat. Jako komunikat uznaje się ciąg znaków otoczonych znakiem podwójnego cudzysłowu.
3. **Rozpoznane tokeny** - jest to tabela pokazująca wyniki działania algorytmu NLP. Prezentuje ona regułę rozbity na pojedyncze tokeny. Przy każdym tokenie wskazana jest logiczna encja, do której przyporządkował go algorytm dokonujący analizy, oraz liczbowe prawdopodobieństwo z jaką nastąpiło dopasowanie. Ostatnia kolumna tej tabeli ma zastosowanie tylko do wybranych bytów logicznych i służy do normalizacji tokenów. Np. tokeny "żówny", "żówna", "jest równy" należą do jednej wspólnej kategorii oznaczającej równość "=", a tokeny "jest większy lub równy", "jest nie mniejszy niż" oznaczają kategorię >="
4. **Rozpoznane parametry wejściowe** - w tabeli tej prezentowane są tokeny uznane jako parametry wejściowe do reguły. Zwykle występują one w wyrażeniach warunkowych. W przypadku gdy warunek reguły skonstruowany jest w taki sposób że następuje porównanie z wartością stałą (np. liczbą lub datą), wtedy taka wartość jest traktowana jako parametr wejściowy reguły z przypisaną wartością domyślną. Warunkiem poprawności analizy reguły jest właściwe przyporządkowanie typów danych do parametrów reguły. Jeśli nie uda się wnioskowanie automatyczne, to informacje te muszą być wprowadzone przez użytkownika w do drugiej kolumny tabeli parametrów.
5. **Rozpoznane wywołania innych reguł** - W tabeli tej prezentowane są odnalezione odwołania do innych reguł. Konieczne jest dokonanie mapowania parametrów wejściowych reguły wołającej i reguły wołanej.

**P**rezentacja wyników następuje po wybraniu reguły z listy w panelu bocznym. Zielony znaczek świadczy o tym że dopełnione zostały wszystkie doprecyzowania i mapowania parametrów, więc aplikacja posiada komplet informacji potrzebnych do wygenerowania kodu.





Rysunek 3: Główne okno aplikacji - sekcja wygenerowanego kodu

Uwaga! Sekcja ta jest dostępna tylko wtedy, gdy wszystkie reguły zaprezentowane na ekranie analizy danych 2 zostaną zwalidowane pozytywnie, czyli aplikacja uzyska wszystkie dane potrzebne do wygenerowania kodu.

## 7 Eksperymenty




Teraz chciałbym pokazać jak działa moja aplikacja. W tym celu spróbuję trochę poeksperymentować z danymi i pokazać to, co do tej pory udało się osiągnąć.

Wyobraźmy sobie sytuację że robimy system rejestrujący badania medyczne wykonane na rzecz pacjenta. Chcielibyśmy się dowiedzieć, czy rejestrowane badanie jest badaniem refundowanym (nasza wyobrażona refundacja dotyczy tylko badań zarejestrowanych w roku 2019). O czasie rejestracji badania mówi zmienna „data\_rejestracji\_badania”. W przypadku gdy badanie nie jest refundowane system ma wyświetlić komunikat informacyjny.

Na początek definicja reguły, niech brzmi ona następująco:

*Jeżeli data\_rejestracji\_badania jest mniejsza niż '01-01-2019' wtedy wyświetl komunikat "Badanie sprzed okresu refundacji".*

Wprowadzam ją do systemu, i wygląda to tak:

Reguły wejściowe		
RS-012	Jeżeli data_rejestracji_badania jest mniejsza niż '01-01-2019' wtedy wyświetl komunikat "Badanie sprzed okresu refundacji".	  

Teraz spójrzmy czego udało się o niej dowiedzieć:

Reguła wejściowa

RS-012

Jeżeli data\_rejestracji\_badania jest mniejsza niż '01-01-2019' wtedy wyświetl komunikat "Badanie sprzed okresu refundacji".

Wyodrębnione komunikaty

KOMUNIKAT1

Badanie sprzed okresu refundacji

Rozpoznane tokeny

Token	Wartość logiczna	Słownie	Prawd.	Kategoria
Jeżeli	SK_SW	Początek warunku	0.191	
data_rejestracji_badania	OP_L	Lewostronny operand porównania	0.184	
jest mniejsza niż	OPR_REL	Operator relacyjny	0.182	<
'01-01-2019'	OP_P	Prawostronny operand porównania	0.193	
wtedy	SK_KW	Koniec warunku	0.184	
wyświetl komunikat	AKCJA	Akcja	0.164	WYSWIETL_KOMUNIKAT
KOMUNIKAT1	AKCJA_PARAMETR	Parametr akcji	0.170	

Rozpoznane parametry wejściowe

Parametr	Typ	Domyślnie
data_rejestracji_badania	<div><div>Data</div><div></div></div>	
const_param2	<div><div>Data</div><div></div></div>	'01-01-2019'

- **Reguła wejściowa** - system jeszcze raz przypomina treść wprowadzonej reguły i prezentuje nadany jej kod, w naszym przypadku RS-012.
- **Wyodrębnione komunikaty** - tu zaprezentowane zostały komunikaty, które udało się wyszukać w regule. Każdy odnaleziony komunikat zostaje wycięty z reguły przed poddaniem jej analizie i zastąpiony swoim reprezentującym go identyfikatorem - w naszym przypadku odnaleziony został jeden komunikat i zastąpiony przez słowo „KOMUNIKAT1”.
- **Rozpoznane tokeny** - najistotniejsza część wyników analizy. Widzimy, że z tą regułą algorytm poradził sobie bezbłędnie. Tokeny zostały prawidłowo skojarzone z ich znaczeniem logicznym. Na chwilę uwagi zasługuje tylko kolumna „Kategoria” . Dotyczy ona tych tokenów, które mogą być określone na wiele sposobów. Musimy przyporządkować im jedną, uniwersalną kategorię. Jako przykład niech posłuży operator mniejszości - może on zostać opisany jako („jest mniejszy niż”, „jest



mniejszy od” „,jest mniejsza od” „...”). Z punktu widzenia aplikacji, wszystkie te wartości reprezentują jedną kategorię, ja nazwałem ją po prostu „<”.

- Rozpoznane parametry wejściowe - w tym miejscu pokazywane są wartości, które uznane zostały za parametry wejściowe reguły. Trafiają tu tokeny, które biorą udział w porównaniach, lub są przekazywane jako parametry wejściowe do innych reguł (tę sytuację pokażę trochę później). W tym przypadku widzimy, że aplikacji udało się prawidłowo dopasować typy danych i określić wartość domyślną jednego z parametrów. Gdyby to się nie udało, czynność tę musi wykonać człowiek. Musi się to stać przed wygenerowaniem kodu.

I na koniec spójrzmy na wygenerowany kod metody walidującej regułę RS-012.




```
fun rs_012(  
    data_rejestracji_badania: LocalDate,  
    const_param2: LocalDate = LocalDate.parse("01-01-2019" ),  
    KOMUNIKAT1: String = "Badanie sprzed okresu refundacji"  
) : wynikDzialaniaReguly {  
    if(data_rejestracji_badania <const_param2) {  
        return komunikat(KOMUNIKAT1 )  
    }  
}
```

Wszystko wygląda ok, więc spróbujmy trochę skomplikować regułę.

Założmy że refundacji podlegają tylko te badania, które zostały wykonane w pierwszym kwartale 2019 roku. Chcielibyśmy zmienić treść reguły w taki sposób, by po stwierdzeniu tego typu sytuacji informowała użytkownika że jego badanie może zostać zrefundowane.

*Jeżeli data\_rejestracji\_badania jest mniejsza niż '01-01-2019' lub data\_rejestracji\_badania nie jest większa niż '01-04-2019' wtedy wyświetl komunikat "Badanie sprzed okresu refundacji".*

Modyfikuję regułę:

Reguły wejściowe		
RS-012	Jeżeli data_rejestracji_badania jest większa niż '01-01-2019' i data_rejestracji_badania jest mniejsza od '01-04-2019' wtedy wyświetl komunikat "Badanie podlega refundacji".	  

I oglądamy wyniki:

Wyodrębnione komunikaty				
KOMUNIKAT1	Badanie podlega refundacji			

Rozpoznane tokeny				
Token	Wartość logiczna	Słownie	Prawd.	Kategoria
Jeżeli	SK_SW	Początek warunku	0.191	
data_rejestracji_badiania	OP_L	Lewostronny operand porównania	0.184	
jest większa niż	OPR_REL	Operator relacyjny	0.181	>
'01-01-2019'	OP_P	Prawostronny operand porównania	0.192	
i	OPR_LOG	Operator logiczny	0.185	&&
data_rejestracji_badiania	OP_L	Lewostronny operand porównania	0.190	
jest mniejsza od	OPR_REL	Operator relacyjny	0.186	<
'01-04-2019'	OP_P	Prawostronny operand porównania	0.187	
wtedy	SK_KW	Koniec warunku	0.188	
wyświetl komunikat	AKCJA	Akcja	0.164	WYSWIETL_KOMUNIKAT
KOMUNIKAT1	AKCJA_PARAMETR	Parametr akcji	0.170	

Rozpoznane parametry wejściowe		
Parametr	Typ	Domyślnie
data_rejestracji_badiania	Data	
const_param2	Data	'01-01-2019'
const_param3	Data	'01-04-2019'

+
-

Istotne zmiany doszły w tabeli rozpoznanych tokenów. Doszedł operator logiczny, oraz druga część warunku. Udało się również prawidłowo przyporządkować operatory do określających je kategorii. Dodatkowo doszedł nowy parametr wejściowy, którego zadaniem jest definiować drugą porównywaną wartość.




Wygenerowany kod również wygląda na poprawny:

```

fun rs_012(
    data_rejestracji_badania: LocalDate,
    const_param2: LocalDate = LocalDate.parse("01-01-2019" ),
    const_param3: LocalDate = LocalDate.parse("01-04-2019" ),
    KOMUNIKAT1: String = "Badanie podlega refundacji"
): WynikDzialaniaReguly {
    if(data_rejestracji_badania >const_param2 && data_rejestracji_badania <const_param3) {
        return Komunikat(KOMUNIKAT1 )
    }
}

```

Ponownie zmodyfikujmy regułę:

Reguły wejściowe		
RS-012	Jeżeli data_rejestracji_badania jest większa niż '01-01-2019' i data_rejestracji_badania jest mniejsza od '01-04-2019' wtedy wyświetl komunikat "Badanie podlega refundacji", w przeciwnym wypadku zgłoś błąd "Badanie nie może zostać zarejestrowane".	  

### Reguła wejściowa

#### RS-012

Jeżeli data\_rejestracji\_badania jest większa niż '01-01-2019' i data\_rejestracji\_badania jest mniejsza od '01-04-2019' wtedy wyświetl komunikat "Badanie podlega refundacji", w przeciwnym wypadku zgłoś błąd "Badanie nie może zostać zarejestrowane" .

### Wyodrębnione komunikaty

KOMUNIKAT1	Badanie podlega refundacji
KOMUNIKAT2	Badanie nie może zostać zarejestrowane

### Rozpoznane tokeny

Token	Wartość logiczna	Słownie	Prawd.	Kategoria
Jeżeli	SK_SW	Początek warunku	0.191	
data_rejestracji_badania	OP_L	Ławostronny operand porównania	0.184	
jest większa niż	OPR_REL	Operator relacyjny	0.181	>
'01-01-2019'	OP_P	Prawostronny operand porównania	0.192	
i	OPR_LOG	Operator logiczny	0.185	& &
data_rejestracji_badania	OP_L	Ławostronny operand porównania	0.190	
jest mniejsza od	OPR_REL	Operator relacyjny	0.186	<
'01-04-2019'	OP_P	Prawostronny operand porównania	0.187	
wtedy	SK_KW	Koniec warunku	0.188	
wyświetl komunikat	AKCJA	Akcja	0.161	WYSWIETL_KOMUNIKAT
KOMUNIKAT1	AKCJA_PARAMETR	Parametr akcji	0.116	
w przeciwnym wypadku	SK_SAN	Początek akcji NIE	0.178	
zgłoś błąd	AKCJA	Akcja	0.194	ZGLOS_BLAD
KOMUNIKAT2	AKCJA_PARAMETR	Parametr akcji	0.178	

```

fun rs_012(
    data_rejestracji_badania: LocalDate,
    const_param2: LocalDate = LocalDate.parse("01-01-2019" ),
    const_param3: LocalDate = LocalDate.parse("01-04-2019" ),
    KOMUNIKAT1: String = "Badanie podlega refundacji",
    KOMUNIKAT2: String = "Badanie nie moÅ¼e zostaÅ zarejestrowane"
): WynikDzialaniaReguly {
    if(data_rejestracji_badania >const_param2 && data_rejestracji_badania <const_param3) {
        return Komunikat(KOMUNIKAT1 )
    }
    else {
        return BladWalidacji(KOMUNIKAT2 )
    }
}

```

## Część III

# Ocena wyników

8    **Prezentacja wyników**

9    **Ocena wyników**