

Realizacja prostego silnika reguł walidacyjnych przy użyciu technik NLP

Mariusz Wójcik

15 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 rozpoczne 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 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.

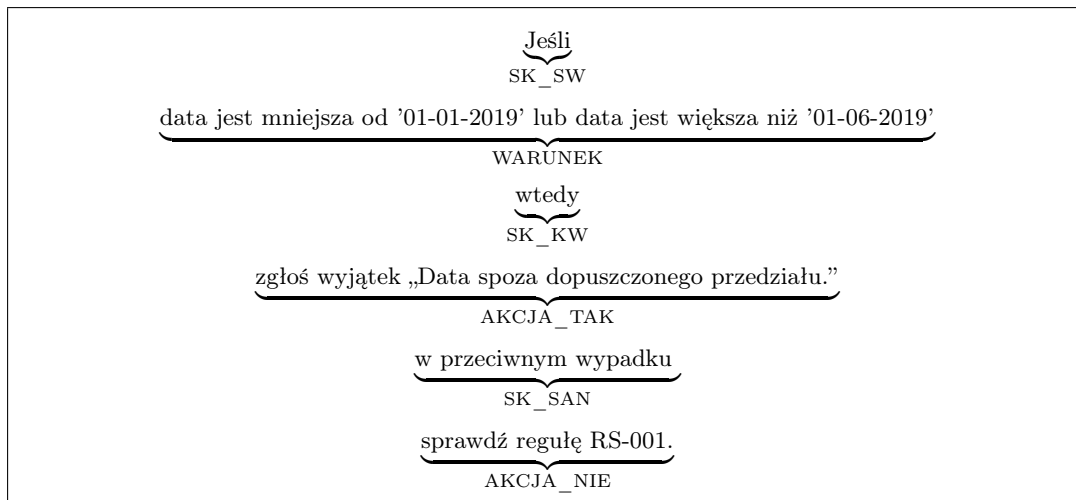
W celu ich wyróżnienia wprowadzam następujące oznaczenia:

1. SK_SW - Start sekcji warunku
2. SK_KW - Koniec sekcji warunku
3. SK_SAN - Start sekcji akcji wykonywanej przy niespełnionym warunku

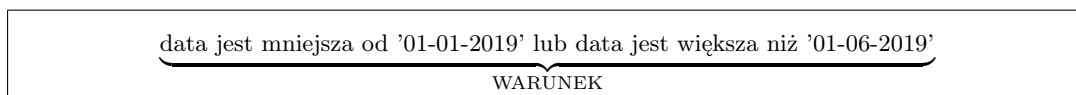
Schemat reguły przyjmuje następującą postać:

$\underbrace{\text{SK_SW}}_{\text{SK_SW}} \underbrace{\text{WARUNKI}}_{\text{WARUNKI}} \underbrace{\text{SK_KW}}_{\text{SK_KW}} \underbrace{\text{AKCJA_TAK}}_{\text{AKCJA_TAK}} \underbrace{(\text{SK_SAN AKCJA_NIE})}_{(\text{SK_SAN AKCJA_NIE})}?$

Rzut oka na przykład:



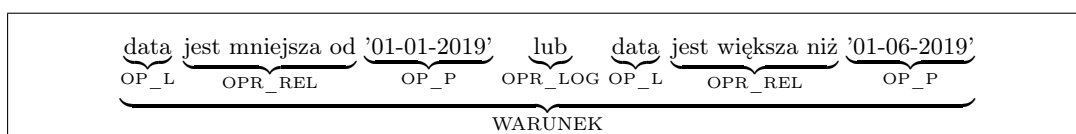
Ponieważ kluczowe jest właściwe rozpoznanie sekcji warunku, chciałbym wyłączyć go przed nawias i przez chwilę skupić się wyłącznie na nim.



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

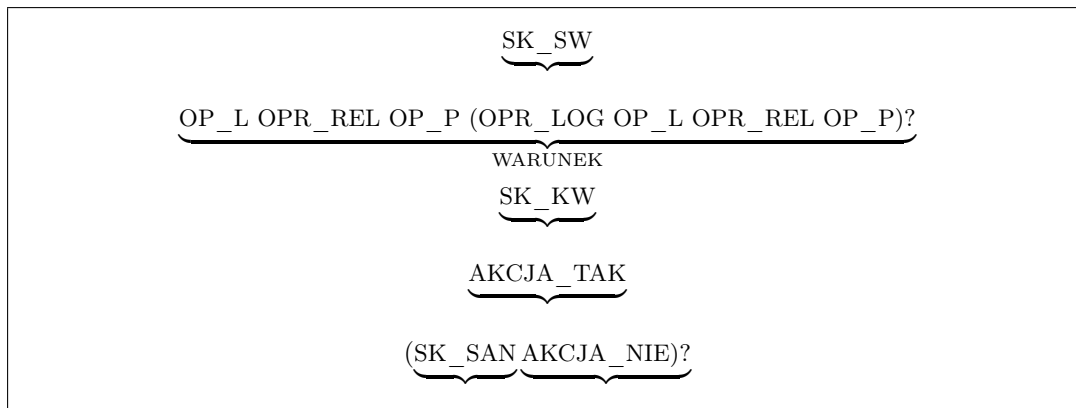
Po podstawieniu, przykładowy warunek można



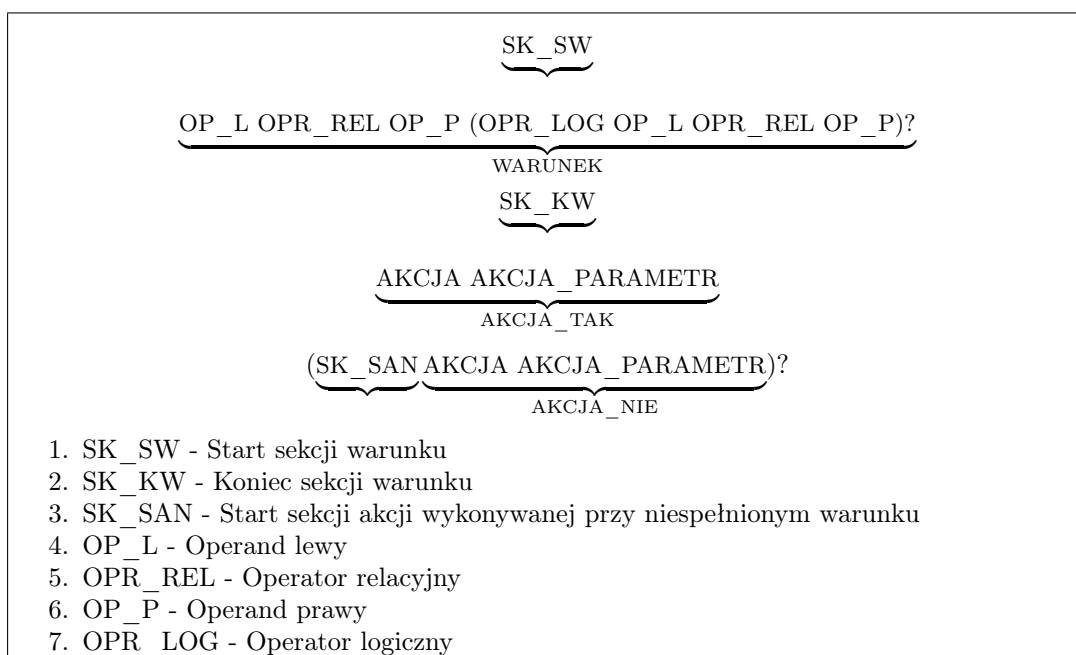
Zatem zapis symboliczny sekcji warunkowej będzie wyglądał następująco:



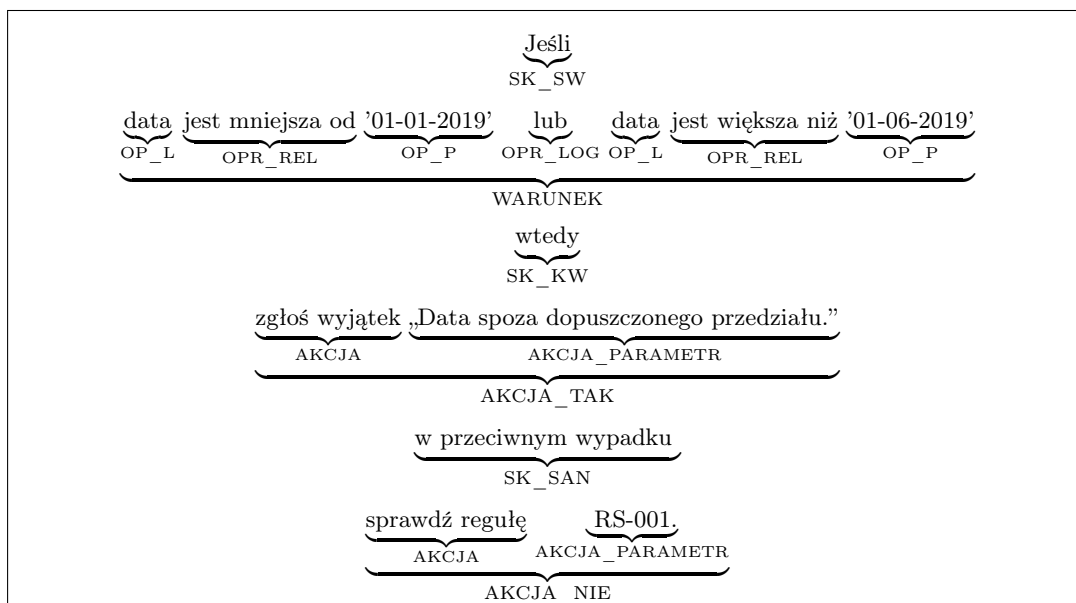
Po dokonaniu podstawienia w sekcji *WARUNKI* abstrakcyjny model reguły przyjmie następującą postać:



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 **Prezentacja wyników**

7 **Ocena wyników**