Semi-supervised neural part-of-speech tagging

Britvić, Tihana

Master's thesis / Diplomski rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:169195

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb







UNIVERSITY OF ZAGREB

FACULTY OF SCIENCE

DEPARTMENT OF MATHEMATICS

Tihana Britvić

SEMI-SUPERVISED NEURAL PART-OF-SPEECH TAGGING

Master thesis

Mentor:
Mladen Vuković

Zagreb, 2018

This Master thesis was defended on _____ in front of the examination Committee formed of:

1. _____, president
2. _____, member
3. _____, member

The Committee evaluated the thesis with _____.

Signatures of Committee Members:

1. _____
2. _____
3. _____

Contents

Contents iii

Introduction 2

1 Data 3
1.1 Word classes . . . 4
1.2 Language Families . . . 5
1.3 Data features . . . 6

2 Method 8
2.1 Feed Forward NN . . . 8
2.2 Recurrent Neural Network . . . 10
2.3 Bidirectional RNNs . . . 12
2.4 Multi-Layer RNNs . . . 14
2.5 Long Term Dependencies Problem . . . 15
2.6 Gated architectures . . . 15
2.7 Tagging with bi-LSTMs . . . 17

3 Experiments 19
3.1 Tag handling . . . 21
3.2 Algorithm Random Tag . . . 22
3.3 The Blank Tag Algorithm . . . 23

4 Results 28
4.1 Random Tag algorithm results . . . 30
4.2 Results of method Blank Tag . . . 31
4.3 Results per Families . . . 33
4.4 Related work . . . 35

5 Conclusion 37

Bibliography **38**

Introduction

In corpus linguistics, part-of-speech tagging (POS tagging) is the process of marking up a word in a text (also called corpus) as corresponding to a particular part of speech, based on both its definition and its context. For example, we can observe the relationship between adjacent and related words in a phrase, sentence, or paragraph. One of the main challenges in POS tagging is **ambiguity** since one word can take several possible parts of speech. Another problem is words or parts of speech which are complex or unspoken. Both of these problems are not rare in natural languages.

Certain languages are called **low-resourced** for a few reasons. One of them is that they are actually low-density languages, meaning they are not spoken by a large number of people in the world. Examples include Inuit or Sindhi. On the other hand, we have technologically low-resource languages. For example, until 2004 Hindi was considered to be a low-resource language since it was really difficult to find Hindi corpora anywhere online. Another example is Urdu, which falls under the same category although its resourcefulness increased over the last few years.

Natural-language processing (NLP) is a field of computer science, artificial intelligence concerned with the interactions between computers and human (natural) languages. Challenges in NLP frequently involve speech recognition, natural-language understanding, and natural-language generation, although the most famous problem is POS tagging.

There are different approaches to this problem, but when the previous work is considered, it all comes down to cross-lingual projections and machine learning. Most of the previous work done in NLP has been limited to training and evaluating on no more than a dozen languages, typically all from the major Indo-European languages (with emphasis on Romance and Germanic families).

In contrast, this paper presents an effort to learn POS taggers for truly low-resource languages, with minimum assumptions about the available language resources. Most low-resource languages are non-Indo-European, and typically, their typological and geographic neighbors have sparse resources as well. Our experiments showed that word dictionaries allow us to create very powerful semi-supervised learning technique. A bidirectional long short-term memory (bi-LSTM) network was created. Bi-LSTMs have recently proven successful for various NLP sequence modeling tasks, although little is known about their

reliance to input representations, target languages, data set size, and label noise. We address these issues and evaluate bi-LSTMs with word and character embeddings for POS tagging.

The obtained results were competitive with approaches that assume the availability of larger volumes of tag dictionaries and also, representation of parallel corpora (cross-lingual methods) or a perfectly tokenized monolingual corpora for our target languages.

I would like to thank professor Željko Agić, my other mentor, for making this paper possible. You are a true inspiration as a lecturer, a mentor, and a friend. Thank you for everything you taught me, for presentations which encouraged me, and for dealing with me during my internship. For this, and everything else, I will forever be grateful.

Chapter 1

Data

For the multilingual experiments, we used the data from the Universal Dependencies project v1.2 (Nivre et al., 2015) (17 POS) with the canonical data splits which we converted to 12 POS using standard mappings which can be found at `https://github.com/slavpetrov/universal-pos-tags`. If there was more than one treebank per language, we use the treebank that has the canonical language name (e.g., Finnish instead of Finnish-FTB). We consider all languages that have at least 60k tokens and are distributed with word forms which resulted in 24 languages. For each language, four different files were provided: train file, test file, dictionary file and dev file.
Each of files is either given or converted into following form:

- each line of a file is structured as "*word*" < *tab* > "*tag*" < \n >,
- all sentences are divided by the blank line.

Train file is the largest file used for the network training. We consider tags from train file as unreliable and we focus on tags from the dictionary file which contains hand tagged words. Dictionary tags are considered the most reliable for POS tagging. Test file is considered to carry "golden" tags, tags used to measure the accuracy of our neural network. Similarly to test file, dev file is used for testing the network after each iteration. Due to accuracy form testing dev file testing, we can easily spot over or underfitting of a network.

Dictionary files

To improve comparison with the related work the dictionaries for nine languages were changed: Danish, German, English Greek, English, Spanish, Italian, Dutch, Portuguese and Swedish. For them, Wiktionary, a freely available, high coverage and constantly growing dictionary for a large number of languages, was used. The Wiktionary dictionaries

with corresponding dictionary mappings are available at: `https://code.google.com/archive/p/wikily-supervised-pos-tagger/downloads/`.

1.1 Word classes

As already mentioned, focus was on tagging words into 12 word classes which are given and explained on a Figure 1.1. Some of word classes are considered as **open**, which means that new words can be added to the class as the need arises. For instance, the class of nouns is potentially infinite, since it is continually being expanded as new scientific discoveries are made, new products are developed, and new ideas are explored.
Examples: *Internet, website, URL, CD-ROM, email, newsgroup, bitmap, modem, multimedia*. [11]. Similary, new verbs have been introduced, i.e., *download, upload, reboot, right-click, double-click*, which makes verbs also an open class. The adjective and adverb classes can also be expanded by the addition of new words, though less prolifically.
On the other hand, new prepositions, determiners, or conjunctions are never invented. Hence, they are called **closed** word classes and are made of finite sets of words which are never expanded. The subclass of pronouns, within the open noun class, is also closed. Words in an open class are known as open-class items and words in a closed class are known as closed-class items.

Tag	Meaning	Example (English)	
ADJ	adjective	new, good, high, special, big, local	OPEN CLASS WORDS
ADV	adverb	really, already, still, early, now	
VERB	verb	is, say, told, given, playing, would	
NOUN	noun	year, home, costs, time, Africa	
ADP	adposition	on, of, at, with, by, into, under	CLOSED CLASS WORDS
CONJ	conjunction	and, or, but, if, while, although	
DET	determiner, article	the, a, some, most, every, no, which	
NUM	numeral	twenty-four, fourth, 1991, 14:24	
PRT	particle	at, on, out, over per, that, up, with	
PRON	pronoun	he, their, her, its, my, I, us	
.	punctuation marks	. , ; !	OTHER
X	other	ersatz, esprit, dunno, gr8, univeristy	

Figure 1.1: List of 12 universal tags with the examples for English. The division of word classes is given in the last column.

1.2 Language Families

According to Ethnologue, there are 7 472 known languages out of which 7 099 of them are living human languages distributed in 141 different language families. A ”living language” is one used as the primary form of communication of a group of people. There are also many dead and extinct languages, as well as some that are still insufficiently studied to be classified or are even unknown outside their respective speech communities.

The most numerous is the family of Niger-Congo languages counting 1 524 languages and approximately 437 000 000 speakers, although the most ”spoken” family is the Sino-Tibetan languages with 1 268 000 000 speakers. A full list can be found in [12].

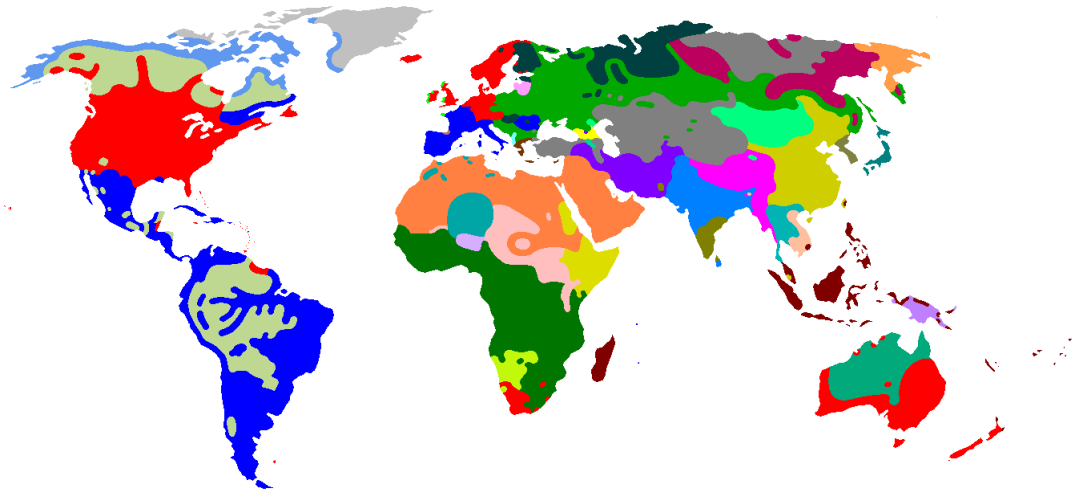


Figure 1.2: Map of language families that show the difference between the languages and how they are spread across the world. Since we are focusing on seven language families and two isolated languages, the legend is intentionally left out. Colors used in all graphs correspond to this figure, meaning: green color represents Slavic, red Germanic, blue Romance, brown Hellenic, orange Semitic, grey Uralic, yellow Isolated languages, and turquoise Indo-Iranian languages.

As already mentioned, our data has 24 different languages which are members of seven different families (Slavic, Germanic, Romance, Hellenic, Semitic, Uralic and Indo-Iranian) and two isolated languages (Basque and Irish). The full list of our languages and their belonging families is presented in the Table 1.1.

Language Families					
Language	Short	Family	Language	Short	Family
Bulgarian	bg	Slavic	Czech	cs	Slavic
Danish	da	Germanic	German	de	Germanic
Greek	el	Hellenic	English	en	Germanic
Spanish	es	Romance	Basque	eu	Language isolate
Persian	fa	Indo-Iranian	Finnish	fi	Uralic
French	fr	Romance	Irish	ga	Language isolate
Ancient Greek	grc	Hellenic	Hebrew	he	Semitic
Hindi	hi	Indo-Iranian	Croatian	hr	Slavic
Hungarian	hu	Uralic	Italian	it	Romance
Latin	la	Romance	Dutch	nl	Germanic
Polish	pl	Slavic	Portuguese	pt	Romance
Romanian	ro	Romance	Swedish	sv	Germanic

Table 1.1: The table of our data languages and their belonging families.

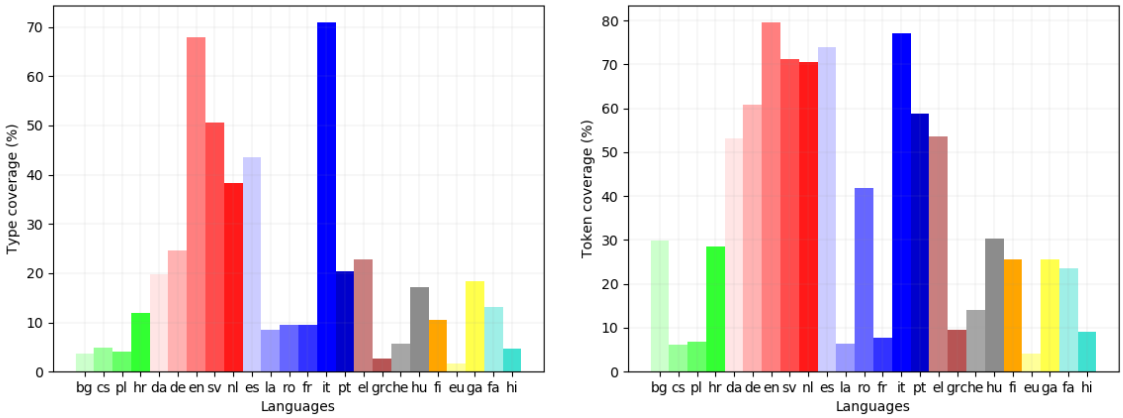
Short columns in Table 1.1 represent names of languages codes by a universal ISO 639-2 Code standard [7].

1.3 Data features

There are two kinds of coverage of interest: **type coverage** and **token coverage**. We define type coverage as the proportion of word types in the corpus that simply appear in the dictionary. Token coverage is defined similarly as the portion of all word tokens in the corpus that appear in the dictionary.

These statistics reflect two aspects of the usefulness of a dictionary that affect learning in different ways: token coverage increases the density of a supervised signal while type coverage increases the diversity of word shape supervision. At one extreme, with 100% word and token coverage, we recover the POS tag disambiguation scenario and, on the other extreme of 0% coverage, we recover the unsupervised POS induction scenario.

Type and token coverages for each of the languages we are using for evaluation which is shown in 1.3. We should accentuate that we used Wiktionary dictionaries for Danish, German, English Greek, English, Spanish Italian, Portuguese and Swedish and Universal Dependencies project v1.2 dictionaries for other languages. On the Figure 1.3 we can see that even for languages whose type coverage is relatively low, such as Greek (el), the token level coverage is still quite good (more than half of the tokens are covered). The reason for this is likely the bias of the contributors towards more frequent words.



(a) Type coverage diagram.

(b) Token coverage diagram.

Figure 1.3: On (a) and (b) diagrams we can see percentages of type and token coverage for each of our 24 languages. We can notice that the Germanic languages have the highest (both) type and token coverage out of all families. The languages are sorted according to the belonging families and colored respectfully by 1.2.

Chapter 2

Method

2.1 Feed Forward NN

In this chapter, the construction of our network will be described. In order to do so, some simpler models of neural networks must be defined. This will enable us to build our final bi-LSTM neural network. Let's start with the simplest neuron network called perceptron.

Definition 1. Let's assume that d_{in}, $d_{out} \in \mathbb{N}$ are fixed numbers, $W \in \mathbb{R}^{d_{in} \times d_{out}}, b \in \mathbb{R}^{d_{out}}$. A function $NN_{Perceptron} : \mathbb{R}^{d_{in}} \longrightarrow \mathbb{R}^{d_{out}}$ defined with:

$$NN_{Perceptron}(x) = xW + b,$$

is called **perceptron**.

Commonly, vector x is called an input vector, vector y is called an output vector, the matrix W is called a weight matrix, and a vector b is called a bias term of a neural network.

The network can be divided into layers, meaning an input vector creates so-called **input layer** and an output vector creates **output layer**. All other layers are considered to be **hidden**.

Going beyond linear functions, a non-linear hidden layer must be introduced. The first step is adding a new hidden layer, resulting in multi-layer perceptron with one hidden layer (the simplest non-linear neural network).

Definition 2. Let's assume $d_{in}, d_{out}, d_1 \in \mathbb{N}$ are fixed numbers, $W_1 \in \mathbb{R}^{d_{in} \times d_1}, b_1 \in \mathbb{R}^{d_1}, W_2 \in \mathbb{R}^{d_1 \times d_{out}}, b_2 \in \mathbb{R}^{d_{out}}, g : \mathbb{R}^{d_1} \longrightarrow \mathbb{R}^{d_1}$ is non-linear function. A function $NN_{MLP1} : \mathbb{R}^{d_{in}} \longrightarrow \mathbb{R}^{d_{out}}$ defined with:

$$NN_{MLP1}(x) = g(xW_1 + b_1)W_2 + b_2,$$

is called **multi-layer perceptron with one hidden-layer** (MLP1).

Same as before, vector x is called an input vector and vector y is called an output vector. W_1 and b_1 are a weight matrix and a bias term for the first linear transformation of

the input, and W_2 and b_2 are a weight matrix and bias term for the second linear function. The function g is applied element-wise and is called an **activation function**. The non-linear activation function g has a crucial role in the network's ability to represent complex functions. Without the non-linearity in g, the neural network can only represent linear transformations of the input. More detailed description of activation functions can be found in [4] (sigmoid function, th, hard-th, and ReLU must be mentioned as the most commonly used activation functions).

Next step in building networks would be adding a new layer of linear-transformations and non-linearities, resulting in an MLP with two hidden-layers.

Definition 3. Let's assume $d_{in}, d_{out}, d_1, d_2 \in \mathbb{N}$ are fixed numbers, $W_1 \in \mathbb{R}^{d_{in} \times d_1}, b_1 \in \mathbb{R}^{d_1}, W_2 \in \mathbb{R}^{d_1 \times d_2}, b_2 \in \mathbb{R}^{d_2}, W_3 \in \mathbb{R}^{d_2 \times d_{out}}, b_3 \in \mathbb{R}^{d_{out}}, g_1 : \mathbb{R}^{d_1} \longrightarrow \mathbb{R}^{d_1}, g_2 : \mathbb{R}^{d_2} \longrightarrow \mathbb{R}^{d_2}$. A function $NN_{MLP2} : \mathbb{R}^{d_{in}} \longrightarrow \mathbb{R}^{d_{out}}$ defined with:

$$NN_{MLP2}(x) = (g_2(g_1(xW_1 + b_1)W_2 + b_2))W_3 + b_3$$

is called **multi-layer perceptron with two hidden-layers** (MLP2).

A graphical representation of MLP2 can be seen on Figure 2.1.

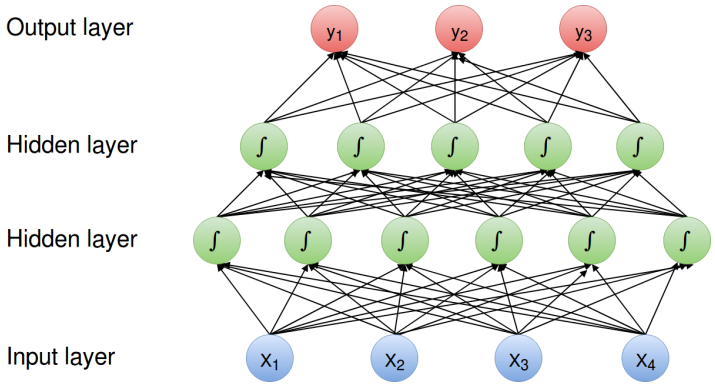


Figure 2.1: Simple sketch of a feed-forward neural network with two hidden layers. Each arrow carries a weight, reflecting its importance (not shown). Weights are forming weight matrices.

Obviously, we can continue building multi-layer perceptron in order to get more and more hidden layers. Networks with multiple hidden layers are called **deep networks**. Deeper networks are usually defined by using intermediary variables for each layer. For

example, in MLP2 one can introduce variables h_1 and h_2 as:

$$\begin{aligned} NN_{MLP2}(x) &= y, \\ y &= h_2 W_3 + b_3, \\ h_2 &= g_2(h_1 W_2 + b_2), \\ h_1 &= g_1(xW_1 + b_1), \end{aligned}$$

where $x, y, b_1, b_2, b_3, W_1, W_2, W_3, g_1$,and g_2 are same as in definition 3.
Naturally follows the definition of a multi-layer perceptron with an arbitrary number of hidden layers.

Definition 4. Let's assume $m, d_{in}, d_{out}, d_1, ..., d_m \in \mathbb{N}$ are fixed numbers, $W_1 \in \mathbb{R}^{d_{in} \times d_1}$, $W_{m+1} \in \mathbb{R}^{d_m \times d_{out}}$, $W_i \in \mathbb{R}^{d_i \times d_j}$, where $j = i+1$, $g_i : \mathbb{R}^{d_i} \longrightarrow \mathbb{R}^{d_i}$, $b_i \in \mathbb{R}^{d_i}$, $\forall i \in \{1, ..., m\}$, $b_{m+1} \in \mathbb{R}^{d_{out}}$. A function $NN_{MLPm} : \mathbb{R}^{d_{in}} \longrightarrow \mathbb{R}^{d_{out}}$ defined with:

$$\begin{aligned} NN_{MLPm}(x) &= y, \\ y &= h_m W_{m+1} + b_{m+1} \\ h_i &= g_i(h_{i-1} W_i + b_i), \forall i \in \{2, ..., m\} \\ h_1 &= g_1(xW_1 + b_1), \end{aligned}$$

is called a **multi-layer perceptron with *m* hidden-layers** (MLPm).

Neural networks output interpretation

Networks with $d_{out} = k > 1$ can be used for **k-class classification**, by associating each dimension with a class, and looking for the dimension with maximal value. Similarly, if the output vector entries are positive and sum to one, the output can be interpreted as a **distribution over class assignments**.

In our tagging problem, a 12-class classification was used ($k = 12$ since there are 12 classes of possible word tags). Meaning, each class of tags was associated with one dimension. The network was designed to give output vector entries as both positive and sum to one, which meant we could interpret output as a distribution over tags.

2.2 Recurrent Neural Network

To go from multi-layer networks to recurrent networks, we need to take advantage of sharing parameters across different parts of a model. Parameter sharing makes it possible to

extend and apply the model to examples of different lengths and generalize across them. That brings us to the introduction of so-called recurrent neural networks (RNN). Unlike multi-layered perceptron which computes over fixed size input vectors, RNN computes over arbitrarily sized inputs of fixed-size vectors. Meaning, instead of input vector $x \in \mathbb{R}^{d_{in}}$ RNN takes a sequence of vectors $x_{1:n} = (x_1, ..., x_n)$, where $x_i \in \mathbb{R}^{d_{in}}, \forall i \in \{1, ..., n\}$, while preserving a same type of output (a vector $y \in \mathbb{R}^{d_{out}}$. With $x_{i:j}$ a sequence of vectors $(x_i, ..., x_j)$ will be denoted in further text.

Definition 5. Let's assume that n, d_{in}, d_{out} are fixed numbers, $x_{1:n} \in \mathbb{R}^{n \times d_{in}}, W_1 \in \mathbb{R}^{d_{in} \times d_{out}}, U, W_2 \in \mathbb{R}^{d_{out} \times d_{out}}, b_1, b_2 \in \mathbb{R}^{d_{out}}, g_1, g_2 : \mathbb{R}^{d_{out}} \longrightarrow \mathbb{R}^{d_{out}}$. A function $RNN : \mathbb{R}^{n \times d_{in}} \longrightarrow \mathbb{R}^{d_{out}}$, defined with $RNN(x_{1:n}) = y_n$, where y_n is defined recursively for $i \in \{1, ..., n\}$ with:

$$y_i = g_2(h_i W_2 + b_2),$$
$$h_i = g_1(x_i W_1 + h_{i-1} U + b_1),$$

we call a **reccurent neural network**.

Definition 6. Let's assume that $n, d_{in}, d_{out} \in \mathbb{N}$ are given. We define a n-sequence of RNN as a function $RNN^* : \mathbb{R}^{n \times d_{in}} \longrightarrow R^{n \times d_{out}}$ with $RNN^*(x_{1:n}) = y_{1:n}$, where $y_i = RNN(x_{1:i}), \forall i \in \{1, ..., n\}$.

Each layer of the RNN representation can be thought of as the state of the computer's memory after executing a set of instructions in parallel. Networks with greater depth can execute more instructions in sequence. Sequential instructions offer great power because later instructions can refer back to the results of earlier instructions. This is why we are inventing a state signal between each layer of an RNN. For each vector x_i in vector sequence $x_{1:n}$ we invent a **state signal** $s_i \in \mathbb{R}$, creating a set of states $S = \{s_1, ..., s_n\}$. With this, RNNs are often denoted a tuple $RNN = (R, O)$ where R stand for a RNN computation given by definition 5. and O is a function $O : S \longrightarrow S' \subset \mathscr{P}(S)$, defined with $O(s_i) = \{s_1, ..., s_{i-1}\}$. On Figure 2.2 the graphical representation can be seen.

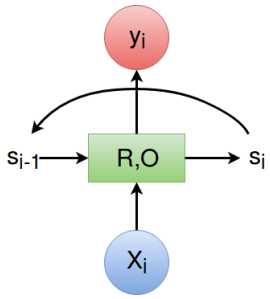


Figure 2.2: Graphical representation of an RNN.

For a finite sized input sequence (i.e. all of our input sequences) the recursion can be unrolled, resulting in the structure in Figure 2.3. Different initializations of R and O

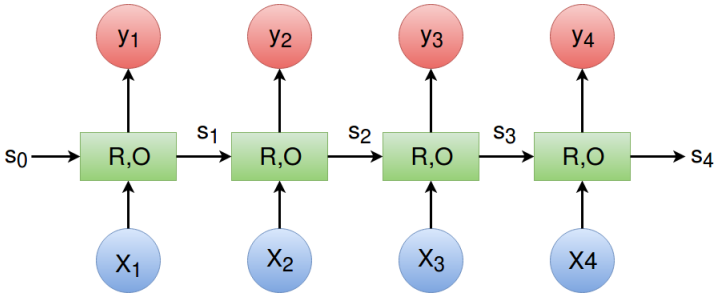


Figure 2.3: Graphical representation of an RNN (unrolled).

will result in different network structures, and will exhibit different properties in terms of their running times and their ability to be trained effectively using gradient-based methods. Thus, s_n and y_n can be thought of as encoding the entire input sequence. The network trains parameters of R and O to be set such that the state conveys useful information for the task we are trying to solve.

2.3 Bidirectional RNNs

A useful elaboration of an RNN is a bidirectional-RNN (bi-RNN) [Graves, 2008]. Consider the task of sequence tagging over a sentence $(x_1, ..., x_n)$. Unlike RNN which allows us to compute a function of the ith word x_i based on the past words $(x_{1:i-1})$, bi-RNN allows us to use both past and the following words $(x_{i+1:n})$ for computation.

Let's suppose RNNs are given, $bi - RNN^f, bi - RNN^b$ with their outputs vectors y_n^f and y_n^b. With $bi - RNN$ we denote a concatenation of vectors y_n^f and y_n^b, marked with $y_n = [y_n^f, y_n^b]$, and we are introducing a notation:

$$bi - RNN = [bi - RNN^f, bi - RNN^b].$$

If we consider an input sequence $x_{1:n}$, the bi-RNN clearly maintains two separate states, s_i^f and s_i^b for each input position $i \in \{1, ..., n\}$. The forward state s_i^f is based on $x_1, x_2, ..., x_i$, while the backward state s_i^b is based on $x_n, x_{n-1}, ..., x_i$. The forward and backward states are generated by two different RNNs ($bi{-}RNN^f$ and $bi{-}RNN^b$). The first $bi{-}RNN^f = (R^f, O^f)$ is fed the input sequence $x_{1:n}$, while $bi - RNN^b = (R^b, O^b)$ is fed the input sequence in reverse. The state representation s_i is then composed of both the forward and backward states. Meaning, the output at position i is based on the concatenation of the two output vectors

$$y_i = [y_i^f, y_i^b] = [O^f(s_i^f), O^b(s_i^b)].$$

In other words, the bi-RNN's encoding y_i of the ith word in a sequence is the concatenation of two RNNs, one reading the sequence from the beginning, and the other reading it from the end. We can calculate a $bi - RNN(x_{1:n}, i)$ as the output vector corresponding to the ith sequence position of $bi - RNN$:

$$bi - RNN(x_{1:n}, i) = y_i = [RNN^f(x_{1:i}), RNN^b(x_{n:i})]$$

The vector y_i can then be used directly for prediction or fed as part of the input to a more complex network. While the two RNNs are run independently of each other, the error gradients at position i will flow both forward and backward through the two RNNs. Feeding the vector y_i through an MLP prior to prediction will further mix the forward and backward signals. Visual representation of the bi-RNN architecture is given in Figure 2.4.

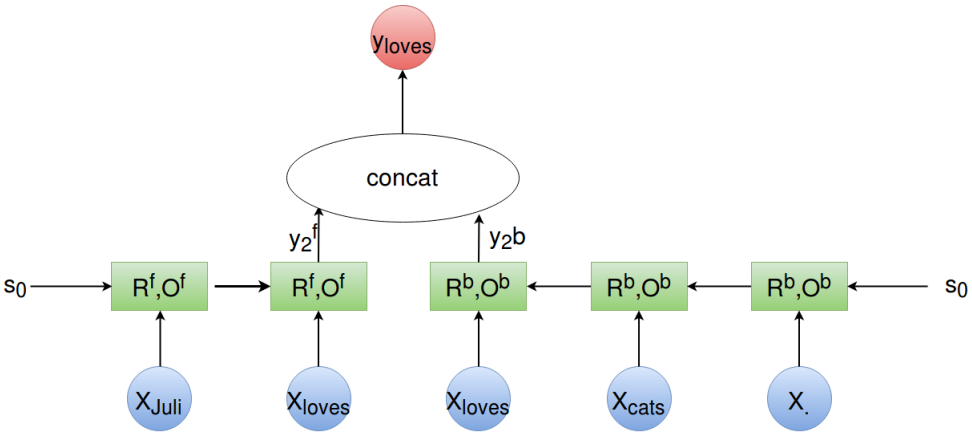


Figure 2.4: Computing the bi-RNN representation of the word jumped in the sentence *Juli loves cats.*.

As in RNN, with $bi - RNN^*(x_{1:n})$ we denote a the sequence of vectors $y_{1:n}$:

$$bi - RNN^*(x_{1:n}) = y_{i:n} = [bi - RNN(x_{1:n}, 1), ..., bi - RNN(x_{1:n}, n)]$$

The n output vectors $y_{i:n}$ can be efficiently computed in linear time by first running the forward and backward RNNs, and then concatenating the relevant outputs. This architecture is depicted in Figure 2.5.

The bi-RNN is very effective for tagging tasks, in which each input vector corresponds to one output vector.

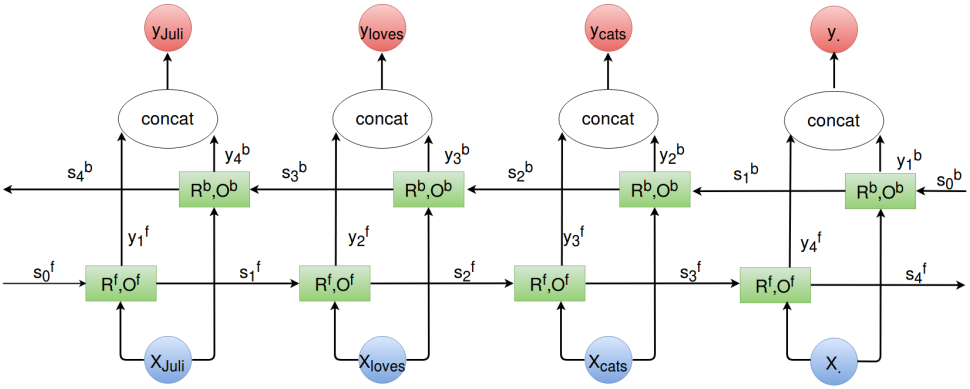


Figure 2.5: Computing the $bi - RNN^*$ for the sentence *Juli loves cats.* .

2.4 Multi-Layer RNNs

RNNs can be stacked in layers, forming a grid [Hihi and Bengio, 1996]. Consider k RNNs: $RNN_1, ..., RNN_k$, where the jth RNN has states $s^j_{1:n}$ and outputs $y^j_{1:n}$. The input for the first RNN are $x_{1:n}$, while the input of the jth RNN ($j \geq 2$) are the outputs of the RNN below it, $y^{j-1}_{1:n}$. The output of the entire formation is the output of the last RNN, $y^k_{1:n}$. Such layered architectures are called **deep RNNs**. A visual representation of a 3-layer RNN is given in Figure 2.6. Let us denote that bi-RNNs can be stacked in a similar way, which has been done in our tagger.

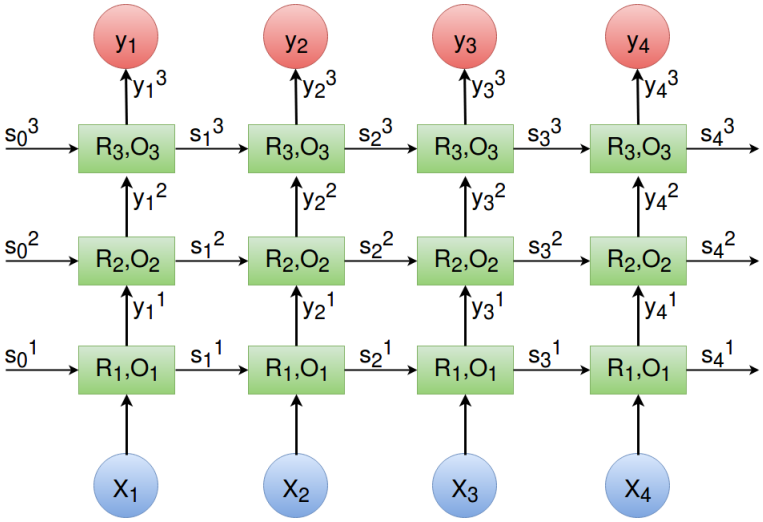


Figure 2.6: A 3-layer RNN architecture for 4-dimensional input vector. This architecture can be applied on our example *Juli loves cats.* for character embedding layer.

2.5 Long Term Dependencies Problem

The idea that RNNs might be able to connect previous information to the present task is extremely appealing. Sometimes, we only need to look at recent information to perform the present task. For example, consider similar language model to our own: a model that is trying to predict the last word based on the previous words in the context. In simple cases, i.e., *The grass is green*, we do not need any further context. It is obvious that the last word would have been *green*. In such cases, where the gap between the relevant information and the place that it is needed is relatively small, RNNs can learn to use the past information. But if we consider an example *I grew up in Italy, which is the reason why I speak fluent Italian.*, in which we are trying to predict a word *Italian*. Recent information suggests that the next word is probably the name of a language, but we need the context of Italy (from further back) to be able to narrow down on exactly which language. From this example, we can see that it is possible for the gap between the relevant information and the point where it is needed to become very large. Unfortunately, as a gap grows, RNN becomes unable to learn to connect the information. Thankfully, LSTMs solve this problem.

2.6 Gated architectures

Considering RNN as a general purpose computing device, where the state s_i represents a finite memory results in following: every time we apply the function R it reads in an input x_{i+1}, reads in the current memory s_i, operates on them (in some way), and writes the result into memory, resulting in a new memory state s_{i+1}. Viewed this way, an apparent problem with the simple RNN architectures because the memory access is not controlled. At each step of the computation, the entire memory state is read, and the entire memory state is written.

The memory access can be controlled by **gates**. A simple example of a gate is a binary vector $g \in \{0,1\}^n, n \in \mathbb{N}$. Such vector can act as a gate for controlling access to n-dimensional vectors, using the element-wise product multiplication of two vectors: $x = u \odot v$ where $x_{[i]} = u_{[i]} \cdot v_{[i]}$ operation $x \odot g$. Let us denote a memory $s \in \mathbb{R}^d$, an input vector $x \in \mathbb{R}^d$, and a gate $g \in \{0,1\}^d$. The computation $s' \longleftarrow g \odot x + (1-g) \odot s$ reads the entries in x that correspond to the 1 values in g, and writes them to the new memory s'. Then, locations that were not read to are copied from the memory s to the new memory s' through the use of the gate $(1-g)$.
The gating mechanism described above can serve as a building block in our RNN: gate vectors can be used to control access to the memory state s_i. However, we are still missing two important (and related) components: the gates should not be static but be controlled by the current memory state and the input, and their behavior should be learned. A solution to the above problem is called a **differentiable gating mechanism**. Changing requirements for

gates and allowing arbitrary real numbers, meaning $g' \in \mathbb{R}^n$, which are then pass through a sigmoid function $\sigma(g')$. This bounds the value in the range (0, 1), with most values near the borders. When using the gate $\sigma(g') \odot x$, indices in x corresponding to near-one values in $\sigma(g')$ are allowed to pass, while those corresponding to near-zero values are blocked. The gate values can then be conditioned on the input and the current memory, and trained jointly as a part of a network.

Long Short-Term Memory neural networks

The Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] was designed to solve the vanishing gradients problem while using the gating mechanism. The LSTM architecture explicitly splits the state vector s_i into two halves, where one half is treated as *memory cells* and the other as *working memory*. The memory cells are designed to preserve the memory, and also the error gradients, across time, and are controlled through differentiable gating components. At each input state, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content in the memory cell should be forgotten.

Definition 9. Let's suppose $n, d_{in}, d_{out} \in N$ are fixed, $x_{1:n} \in \mathbb{R}^{n \times d_{in}}, S = \{s_0, s_1, ..., s_n\} \subset \mathbb{R}, g^q, g^\sigma, g^f, g^s : \mathbb{R}^{d_{out}} \longrightarrow \mathbb{R}^{d_{out}}$, $U^q, U^\sigma, U^f, U^s \in \mathbb{R}^{d_{in} \times d_{out}}, W^q, W^\sigma, W^f, W^s \in \mathbb{R}^{d_{out} \times d_{out}}$. Neural network with a **LSTM architecture** is a function $NN_{LSTM} : \mathbb{R}^{n \times d_{in}} \times S \longrightarrow \mathbb{R}^{d_{out}} \times S$, defined with $NN_{LSTM}(x_{1:n}, s_0) = (y_n, s_n)$, where $y_n = h_n$ and h_n and s_i are defined recursively for $i \in 1, ..., n$:

$$
\begin{aligned}
h_i &= th(s_i) q_i, \\
q_i &= g^q(b_i^q + x_i U_i^q + h_{i-1} W^q), \\
s_i &= f_i s_{i-1} + \sigma_i g^s(b_i^s + x_i U^s + h_{i-1} W^s), \\
\sigma_i &= g^\sigma(b_i^\sigma + x_i U^\sigma + h_{i-1} W^\sigma), \\
f_i &= g^f(b_i^f + x_i U^f + h_{i-1} W^f).
\end{aligned}
$$

The state at time j is composed of two vectors, s_j and h_j, where s_j is the state signal representing the memory component and h_j is the hidden state component. Function f is called **forget gate** which controls what to forget, function σ is called **input gate** and is controlling the input and function q is called **output gate** and in controlling output. The gate values are computed based on linear combinations of the current input x_j and the previous state h_{j-1}, passed through a sigmoid activation function.

2.7 Tagging with bi-LSTMs

Our model for POS tagging refers to two related bi-LSTM architectures which we call the **context bi-RNN** ($bi - RNN_{ctx}$) and the **sequence bi-RNN** ($bi - RNN_{seq}$). In a $bi - RNN_{seq}$, the input is a sequence of vectors $x_{1:n}$ and the output is a concatenation of a forward and backward RNN each reading the sequence in a different direction:

$$y = bi - RNN_{seq}(x_{1:n}) = [RNN^f(x_{1:n}), RNN^b(x_{n:1})]$$

In a $bi - RNN_{ctx}$, we get an additional input i indicating a sequence position, and the resulting vectors y_i result from concatenating the RNN encodings up to i:

$$y_i = bi - RNN_{ctx}(x_{1:n}, i) = RNN^f(x_{1:i}) \circ RNN^b(x_{i:1})$$

Thus, the state vector y_i in our bi-RNN encodes an information at position i and its entire sequential context. Another view of the $bi - RNN_{ctx}$ is of taking a sequence $x_{1:n}$ and returning the corresponding sequence of state vectors $y_{1:n}$.

Our basic bi-LSTM tagging model is a $bi - LSTM_{ctx}$ that takes word embedding vector w as input. We incorporate subtoken information using a hierarchical bi-LSTM architecture (Ling et al., 2015; Ballesteros et al., 2015). The subtoken level computed (character vectors c) embeddings of words using a $bi - LSTM_{seq}$ at the lower level. This representation is then concatenated with the (learned) word embeddings vector w which forms the input to the $bi - LSTM_{ctx}$ at the next layer. This model, illustrated in 2.7 (lower part in left figure), is inspired by Ballesteros et al. (2015). Models in which only subtoken information was kept were also being tested. In our novel model, Figure 2.7 left, we train the bi-LSTM tagger to predict both the tags of the sequence, as well as a label that represents the log frequency of the token as estimated from the training data.
Our combined cross-entropy loss is now:

$$L(\widehat{y_t}, y_t) + L(\widehat{y_a}, y_a), \tag{2.1}$$

where t stands for a POS tag and a is the log frequency label. We calculated the log frequency a as following:

$$a = int(log(freq_{train_file}(w)) \tag{2.2}$$

Combining this log frequency objective with the tagging task can be seen as an instance of multi-task learning in which the labels are predicted jointly. The idea behind this model is to make the representation predictive for frequency, which encourages the model to not share representations between common and rare words, thus benefiting the handling of rare tokens.

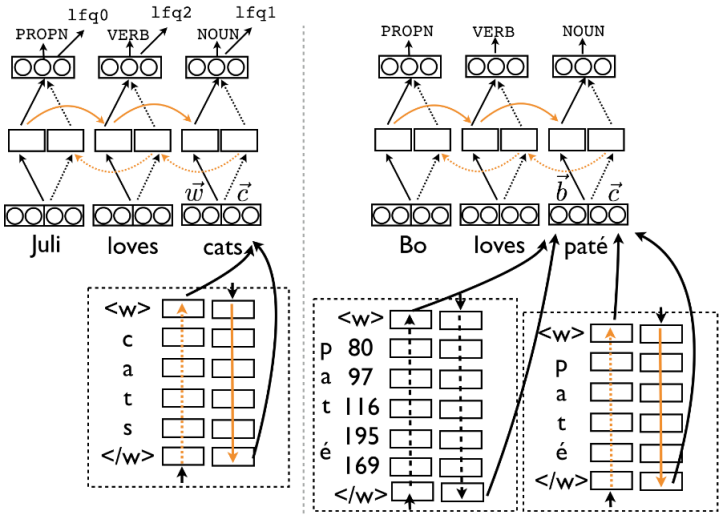


Figure 2.7: Right: bi-LSTM, illustrated with character and word embeddings. Left: *bi – LSTM$_{freq}$*, a multi-task bi-LSTM that predicts at every time step the tag and the frequency class for the token.

Chapter 3

Experiments

All bi-LSTM models were implemented in CNN/pycnn neural network library. For all models we use the same hyperparameters, which were set in English dev, i.e., SGD training with cross-entropy loss, no mini-batches, 20 epochs, default learning rate (0.1), 128 dimensions for word embeddings, 100 for character and byte embeddings, 100 hidden states and Gaussian noise with $\sigma = 0.2$. Embeddings are not initialized with pre-trained embeddings, except when reported otherwise. To simplify, our LSTMs was able to calculate run the data with a possibility of:

- including or excluding character embedding,
- including or excluding word embedding,
- including or excluding auxiliary task,

which led up to $3 \times 2^3 = 24$ combinations for each of 24 languages. We should mention that each experiment was done three times, hence all the results given were a mean out of them.

Character embedding

In a character embedding model, the vector for a word is constructed from the character *n* grams that composed it. Since character *n* grams are shared across words, these models do better than word embedding models for out of vocabulary words - they can generate an embedding for an OOV word. Word embedding models (i.e. word2vec) cannot since they treat a word atomically. Character embedding models tend to do better than word embedding models, for words that occur infrequently, since the character *n* grams that are shared across words can still learn good embedding.

Word embedding

A popular notion in NLP states *"A word is characterized by the company it keeps!"* (Firth). This is exactly what the word embedding means, or more precisely said: word embedding is the collective name for a set of language modeling and feature learning techniques in NLP where words or phrases from the vocabulary are mapped to vectors of real numbers. Conceptually it involves a mathematical embedding from a space with one dimension per word to a continuous vector space with much lower dimension. Methods to generate this mapping include neural networks, dimensionality reduction on the word co-occurrence matrix, probabilistic models, and explicit representation in terms of the context in which words appear. Related studies have shown that when used as the underlying input, word embeddings boost the neural network performance.

In linguistics, word embeddings aim to quantify and categorize semantic similarities between linguistic items based on their distributional properties in large samples of language data. We should mention the most famous word embeddings tool, **word2vec**, which is basically a group of related models used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec was designed by Mikolov in Google (2013).
We must emphasize that when used, word embeddings were off-the-shelf Polyglot embeddings (Al-Rfou et al.,2013) which can be found at `www.let.rug.nl/bplank/bilty/embeds.tar.gz`.

Auxiliary task

In Machine Learning, we typically care about optimizing for a particular metric, whether this is a score on a certain benchmark. In order to do so, we generally train a single model or an ensemble of models to perform our desired task. We then fine-tune and tweak these models until their performance no longer increases. While we can generally achieve acceptable performance this way, by being laser-focused on our single task, we ignore information that might help us do even better on the metric we care about. Specifically, this information comes from the training signals of related tasks. By sharing representations between related tasks, we can enable our model to generalize better on our original task. This approach is called **Multi-Task Learning** (MTL). MLT learning has been used successfully across all applications of machine learning, from natural language processing and speech recognition to computer vision and drug discovery. MTL comes in many guises: joint learning, learning to learn, and learning with auxiliary tasks are only some names that have been used to refer to it. We will be using the last method. As mentioned in 2.7 our bi-LSTM was designed as a learning with the auxiliary task. We at every time step predict the tag and the frequency class for the token using hard parameter sharing, which resulted in calculating loss as described in (2.1).

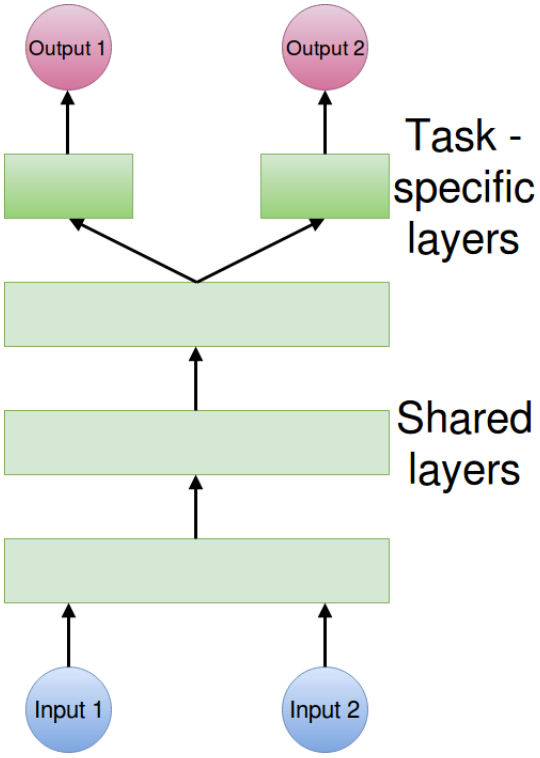


Figure 3.1: We used model of a hard parameter sharing for our MLT part (which greatly reduces the risk of overfitting).

For each language, the auxiliary task file was constructed. As all other files, it was structured as "*word*" < *tab* > "*frequency class*" < \n >, where the frequency class was calculated using (2.2).

3.1 Tag handling

As we focused on low-resource languages using primarily dictionaries as our tool we had to create an algorithm for undeclared words (words not tagged by the dictionary or wrongly tagged words). No restrictions were made on words outside the dictionary, hence we needed to handle three situations:

1. a word is in a dictionary with the corresponding tag
2. a word is in a dictionary but the corresponding tag is not allowed by the dictionary
3. a word is not in a dictionary

The last two cases were our main problem, so we designed two different solutions:

1. Algorithm Random Tag
2. Algorithm Blank Tag

As illustrated in the Figure 3.2, this was done during the so-called preparation phase, meaning we recreated our train data by assigning new tags provided by our two algorithms.

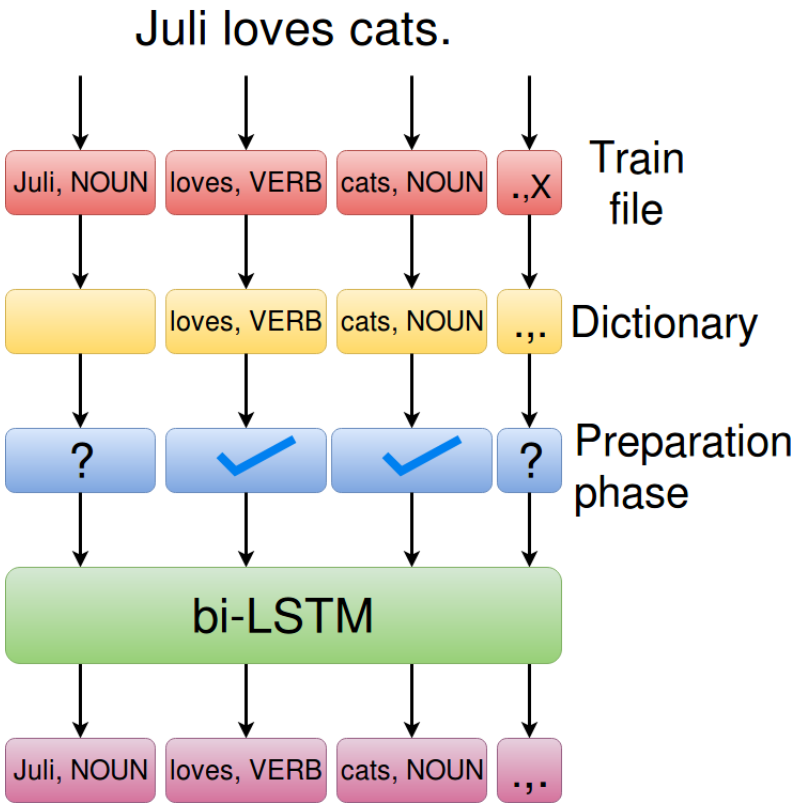


Figure 3.2: Sketch of a process for our example *Juli loves cats.*. Our tag assignment a part of the preparation phase after which the data is used as an input for our bi-LSTM neural network.

3.2 Algorithm Random Tag

First idea was to create an algorithm using a random tag assignment. This meant for each word not correctly tagged we assign a random tag (out of 12 possible tags).

```
for (word, tag) from train file do
    if word in dictionary then
        if tag as in dictionary then
            pass;
        end
        else
            assign word random tag;
        end
    end
    else
        assign word random tag;
    end
end
```

Algorithm 1: Algorithm Random Tag

After reviewed results in several languages, our assumptions were confirmed: "The words belonging to the closed class are in most cases correctly tagged by the tagger". Hence, if we assume that words from the dictionary are correctly tagged, and the data is big enough that the tagger can learn with the high accuracy to tag a word from the closed class, the advantage should be given to tags from open class (while randomly assigning the tags).
To simplify, the random tag was chosen with a probability distribution of 1/4 for open class tags and minimum non-negative float number for the others.

3.3 The Blank Tag Algorithm

The second idea was to assign a new tag (blank tag, marked with *O*) instead of assigning a random tag to undeclared words. This now meant there were 13 possible tags (instead of initial 12). This is how the second algorithm was created:

```
for (word, tag) from train file do
    if word in dictionary then
        if tag as in dictionary then
            pass;
        end
        else
            assign word O tag;
        end
    end
    else
        assign word O tag;
    end
end
```

Algorithm 2: Algorithm Blank Tag

The input of the neural network is a whole sentence and the output is a sequence of corresponding tags, which means that regardless the train data manipulations, the output still has to contain only tags provided by the Figure 1.1. But how to choose these tags? For each word in the sentence, our bi-LSTM gave us a list of 13 values whose sum adds up to 1, meaning they are probabilities that a word should have a corresponding tag. Now, we simply take a tag with:

1. the best probability when it is not equal to the *O* tag
2. the second best probability when it is equal to the *O* tag

Selection of tags can be done in several ways out of which the softmax function is the most commonly used. In our paper, both sotfmax function and approach using Viterbi algorithm were used and results were compared.

Softmax function

Definition 10. Let's suppose we $k \in \mathbb{N}$ is fixed. The softmax function, or normalized exponential function, is a function $softmax : \mathbb{R}^k \longrightarrow [0,1]^k$, defined with:

$$softmax(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{k} e^{x_i}}, \ \forall j = 1, ..., k.$$

In NLP, the output of the softmax function can be used to represent a categorical distribution over classes of tags. If we set k to 13, we can interpret the probability distribution over our 13 possible tags. The most likely tag can then be easily chosen as the maximum element of the outcome of the softmax function. To recall, we need to calculate the most

probable tag out of our initial 12 tags using the probability distribution over the 13 tags, so the final tag is chosen to correspond to:

- the maximum of the probability distribution, if the maximum corresponds to one of 12 initial tags
- the second highest probability of the distribution, if the maximum corresponds to the *O* tag

Viterbi algorithm

Viterbi algorithm is popularly used for finding the most likely sequence of hidden states — called the **Viterbi path** - that results in a sequence of observed events. In many NLP problems, we would like to model pairs of sequences, and our part-of-speech tagging is one of them. The tag sequence is the same length as the input sentence and therefore specifies a single tag for each word in the sentence.

More thoroughly explained: Viterbi algorithm takes a sequence of arbitrary length $n \in \mathbb{N}, x_{1:n} = (x_1, ..., x_n)$, where $x_i \in W = \{w_1, ..., w_s\}$ and generates a path $y_{1:n} = (y_1, ..., y_n)$, where $y_i \in \{1, ..., k\}^n, k \in \mathbb{N}$. In our tagging problem, the sequence $x_{1:n}$ represents a sequence of words, a sentence. Our set W is simply a set of all words appearing in train file hence $s = |W|$. For a sequence $x_{1:n}$, Viterbi algorithm will generate a path $y_{1:n}$ where $y_i \in \{1, ..., k\}^n$ for $k = 13$ represents a sequence of tags.
Beside an input vector, the Viterbi algorithm takes in two matrices called **transition and emission** matrices:

- a transition matrix $A \in \mathbb{R}^{13\times13}$ stores the probabilities of one tag transiting to another, meaning $A_{i,j}$ is a probability that tag_i is followed by tag_j,
- an emission matrix $B \in \mathbb{R}^{13\times12}$ stores the probabilities of from one tag emitting another, meaning $B_{i,j}$ is a probability that tag_i is emitting by tag_j.

Together with A and B, Viterbi algorithm takes in another matrix called **initialization matrix** $\Pi \in \mathbb{R}^{n\times13}$.
During computation, two additional matrices $T_1, T_2 \in \mathbb{R}^{13\times n}$ are constructed:

- each element $T_1[i, j]$ stores the probability of the most likely path so far $\hat{X} = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_j)$, where $\hat{x}_i = tag_j$,
- each element $T_2[i, j]$ stores $\hat{x}_{j-1}$ of the most likely path so far $\hat{X} = (\hat{x}_1, \hat{x}_2, ..., \hat{x}_j)$, where $\hat{x}_i = tag_j$.

Matrices T_1 and T_2 are initialized as zeros, and filled during computations as following:

- $T_1[i, j] = \max_l(T_1[l, j-1] \cdot A_{l,i} \cdot B_{i,x_j})$
- $T_2[i, j] = \operatorname{argmax}_l(T_1[l, j-1] \cdot A_{l,i} \cdot B_{i,x_j})$

Viterbi algorithm can be written with pseudo code:

Data: x, A, B, Π
Result: y
for $i \in \{1, ..., 13\}$ **do**
  $T_1[i, 1] \longleftarrow \Pi_i B_{i,x_i}$
  $T_2[i, 1] \longleftarrow 0$
end
for $i \in \{1, ..., n\}$ **do**
  for $j \in \{1, ..., 13\}$ **do**
    $T_1[i, 1] \longleftarrow \max_l(T_1[l, j-1] \cdot A_{l,i} \cdot B_{i,x_j}$
    $T_2[i, 1] \longleftarrow \operatorname{argmax}_l(T_1[l, j-1] \cdot A_{l,i} \cdot B_{i,x_j})$
  end
end
$z_n \longleftarrow \operatorname{argmax}_l(T_1[l, n])$
$y_n \longleftarrow tag_{z_n}$
for $i \longrightarrow n, n-1, ..., 2$ **do**
  $z_{i-1} \longleftarrow T_2[z_i, i]$
  $y_{i-1} \longleftarrow tag_{z_{i-1}}$
end

Algorithm 3: Viterbi algorithm

Our emission matrix was simply created from outputs of our bi-LSTM. To recall, for each word in a sentence 13 probabilities are given as an output. Transition matrix was created by counting pairs $(tag_i, tag_j), \forall i, j \in \{1, ..., 13\}$ in train file, and then divided by sum of the row. 3.3 we can see the example of how the Viterbi algorithm works for POS tagging.

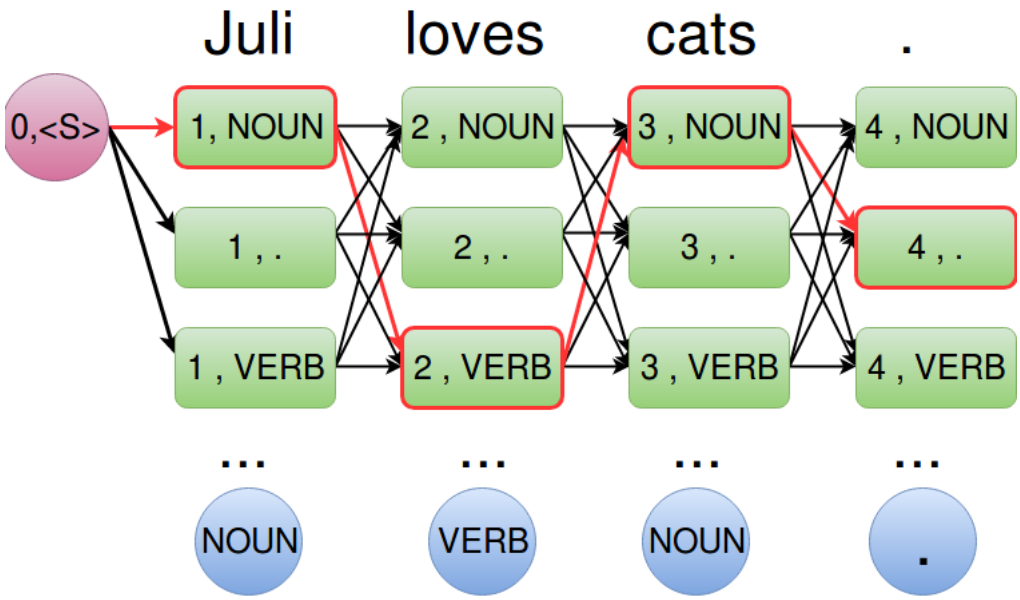


Figure 3.3: Graphical representation of Viterbi algorithm for POS tagging. If the input in Viterbi algorithm is a sentence *Juli loves cats.* than the output should be *NOUN VERB NOUN . .* Arrows represent transitions between states and as such they have weights equal to transition probabilities.

Chapter 4

Results

Since our work was based on low-resource languages we needed a neural network which would have an exponential dependency on the accuracy of network tagging and a coverage of train file with the dictionary. We succeeded, and as can be seen on a Figure 4.1, we have an exponential growth for all cases in each experiment. The dependency test was done by creating five new dictionaries from Danish train file with coverages of 10, 25, 50, 75 and 100 %. These results were taken for the first version of the network, algorithm Random

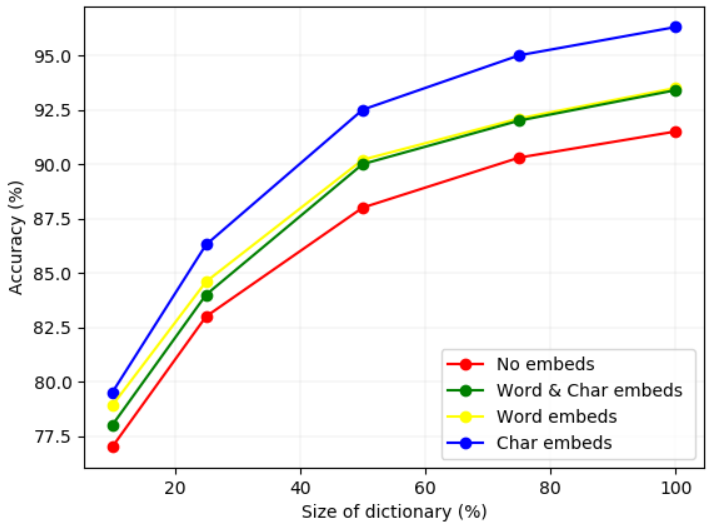


Figure 4.1: Dependency between the accuracy of network tagging and a coverage of train file with the dictionary for Danish.

Tag (and without any additional rules). We should mention that with the improvements, the growth rate of the curve increased.
Results will be given for each method in a separate table and compared later. The structure of a bi-LSTM version can be given by a triple $(w, a, c) \in \{0, 1\}^3$ where w stands for word embeddings, a for the auxiliary task, and c for character embeddings. If were are using one of the possibilities w, a, or c, its place will be marked with 1, and with 0 otherwise.
Since Polyglot word embeddings (see page 20) were not provided Greek, Ancient Greek, Hungarian, Latin, and Romanian, we have not run tests for including them and as such their results will be marked with ✗ in the tables.

4.1 Random Tag algorithm results

Algorithm Random Tag								
ISO	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)	(1, 1, 0)	(1, 1, 1)
bg	53.81	41.05	53.50	61.73	53.46	56.66	47.07	**61.88**
cs	46.22	31.40	28.47	30.11	34.25	25.95	**49.45**	45.64
da	75.31	72.44	75.11	71.27	**76.36**	74.87	69.08	67.69
de	78.63	**81.71**	78.35	80.38	77.13	74.89	76.93	74.61
el	61.38	57.84	60.88	**66.26**	✗	✗	✗	✗
en	85.15	86.71	84.86	85.19	85.00	**86.94**	84.75	85.42
es	81.10	81.10	83.17	82.96	**83.83**	81.36	82.44	81.17
eu	32.20	21.68	23.28	27.43	46.59	42.79	**47.21**	21.69
fa	51.02	41.48	26.35	33.10	46.07	41.45	41.66	**52.94**
fi	36.70	33.45	57.24	36.82	57.55	38.22	**62.89**	62.40
fr	21.14	18.29	20.18	35.70	35.78	31.85	32.46	**35.85**
ga	**49.42**	31.93	21.14	22.07	49.31	48.48	45.29	45.85
grc	27.43	**35.90**	25.01	32.69	✗	✗	✗	✗
he	27.73	25.90	27.84	26.00	32.47	30.73	31.52	**33.21**
hi	14.69	38.33	10.52	9.60	18.06	**38.74**	13.86	15.47
hr	45.55	48.03	**58.35**	58.07	57.78	52.34	46.55	38.93
hu	45.03	45.77	**49.48**	41.36	✗	✗	✗	✗
it	81.08	81.77	81.47	80.67	81.75	80.99	81.22	**84.74**
la	25.93	24.28	**39.10**	39.01	✗	✗	✗	✗
nl	83.40	81.34	83.22	**84.07**	83.62	80.07	83.63	81.41
pl	35.20	28.89	35.22	35.27	**48.79**	28.97	29.59	24.10
pt	71.18	**76.89**	70.42	60.31	64.28	65.89	67.42	62.47
ro	62.78	**74.06**	59.88	69.96	✗	✗	✗	✗
sv	83.17	**86.24**	83.59	82.95	84.91	85.93	85.18	85.21

Table 4.1: The table of results for algorithm Random Tag.

From the Table 4.1 we can see there two configurations gave the best results:

1. configuration with only character embedding included
2. configuration with character embedding, auxiliary task and word embeddings included

If we carefully look at the results, we can conclude that method is not really reliable since the difference in results are big. Also, one has to notice that for train files that have low dictionary coverage, the oscillations between a tagging accuracies for each run are big.

Hence, we can say the lower the percentage of dictionary coverage of a training file, the bigger the oscillations between results of each run. The biggest difference is definitely for Finnish (29,44%).

4.2 Results of method Blank Tag

Method Blank Tag with Softmax

Method Blank Tag with Softmax								
ISO	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)	(1, 1, 0)	(1, 1, 1)
bg	67.65	65.81	67.26	67.65	**73.87**	69.74	71.82	67.32
cs	47.13	49.33	42.13	46.67	52.05	63.26	67.01	**68.92**
da	78.12	79.85	77.35	78.75	80.69	**82.08**	80.86	81.63
de	80.56	85.24	80.79	83.26	83.43	85.31	83.26	**85.57**
el	70.23	72.14	71.71	**72.96**	✗	✗	✗	✗
en	85.47	87.47	86.07	86.98	87.07	**88.29**	87.30	88.11
es	85.87	87.35	85.95	87.30	87.27	87.14	86.76	**87.50**
eu	38.61	52.21	40.37	50.93	52.41	39.92	**54.75**	45.84
fa	51.22	52.94	50.25	52.39	57.38	**61.00**	54.44	59.63
fi	**59.89**	50.71	58.70	49.21	58.67	52.65	53.42	52.90
fr	39.70	42.47	40.31	42.57	43.33	41.99	**44.38**	42.41
ga	50.33	35.98	50.41	49.41	59.59	51.45	**60.53**	50.85
grc	37.02	**37.10**	36.85	36.96	✗	✗	✗	✗
he	31.66	39.12	40.96	39.78	42.81	43.52	45.61	**45.74**
hi	36.95	36.65	20.06	37.58	40.30	**48.10**	39.08	42.94
hr	60.89	**64.79**	59.41	55.56	61.79	61.14	58.12	59.92
hu	54.13	62.96	52.44	**63.00**	✗	✗	✗	✗
it	82.52	85.58	83.27	85.39	84.21	85.35	84.74	**85.73**
la	39.05	41.54	**41.65**	35.54	✗	✗	✗	✗
nl	85.21	88.67	86.04	**89.66**	86.62	85.80	88.89	88.75
pl	46.08	49.14	46.91	48.49	53.09	**54.86**	54.71	54.61
pt	71.69	**79.94**	73.57	79.78	77.44	79.31	74.39	78.37
ro	73.25	72.86	70.50	**75.72**	✗	✗	✗	✗
sv	85.68	87.73	85.56	88.37	87.26	**88.73**	87.27	88.02

Table 4.2: The table of results for Blank Tag Method with Softmax function.

From the table 4.2 we can see that the best configuration was different from the best configuration of algorithm Random Tag. Configuration using word embeddings and character

embeddings and without auxiliary task gave the best results. Almost all results were improved (except for Finnish) and the the differences between results of each run have shorten and stabilized (9.16% on average).

Method Blank Tag with Viterbi algorithm

Method Blank Tag with Viterbi								
ISO	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)	(1, 1, 0)	(1, 1, 1)
bg	65.67	65.53	65.56	66.33	68.22	**72.51**	70.61	70.29
cs	48.01	50.43	48.55	42.80	58.37	61.91	68.16	**68.83**
da	76.09	76.88	76.62	77.79	79.49	78.93	79.56	**81.61**
de	80.33	85.74	80.73	87.04	85.27	87.41	84.47	**88.29**
el	75.94	77.46	75.33	**77.87**	✗	✗	✗	✗
en	85.53	87.29	84.43	86.45	**87.91**	87.43	87.47	87.67
es	83.16	83.70	82.82	85.00	84.98	85.60	85.91	**86.78**
eu	43.67	48.06	43.22	30.32	**52.63**	47.17	50.90	44.33
fa	52.56	52.41	51.23	44.40	56.42	**59.14**	59.11	56.21
fi	62.35	47.14	**62.44**	48.90	60.99	57.65	57.89	52.38
fr	39.65	44.19	40.02	41.03	45.33	**45.67**	40.86	42.27
ga	50.02	49.31	49.96	48.96	57.43	**60.11**	58.61	58.51
grc	36.56	37.27	37.62	**37.74**	✗	✗	✗	✗
he	40.66	39.79	37.99	39.86	44.32	44.70	42.98	**45.14**
hi	38.46	39.48	23.78	39.35	38.50	42.27	**44.50**	43.93
hr	60.83	52.05	60.17	57.92	61.90	60.50	59.64	**62.13**
hu	57.17	**63.70**	50.77	61.15	✗	✗	✗	✗
it	81.93	83.99	80.93	84.48	84.14	86.05	83.14	**86.85**
la	40.46	24.55	40.91	**44.78**	✗	✗	✗	✗
nl	86.10	**89.71**	86.37	89.12	87.78	89.44	86.54	88.96
pl	47.08	48.07	47.52	49.10	**56.06**	53.48	54.72	55.34
pt	82.32	86.04	83.21	85.94	83.47	**88.46**	86.64	88.19
ro	73.64	75.01	69.32	**76.20**	✗	✗	✗	✗
sv	80.62	83.45	82.55	81.67	82.94	**84.72**	82.39	83.66

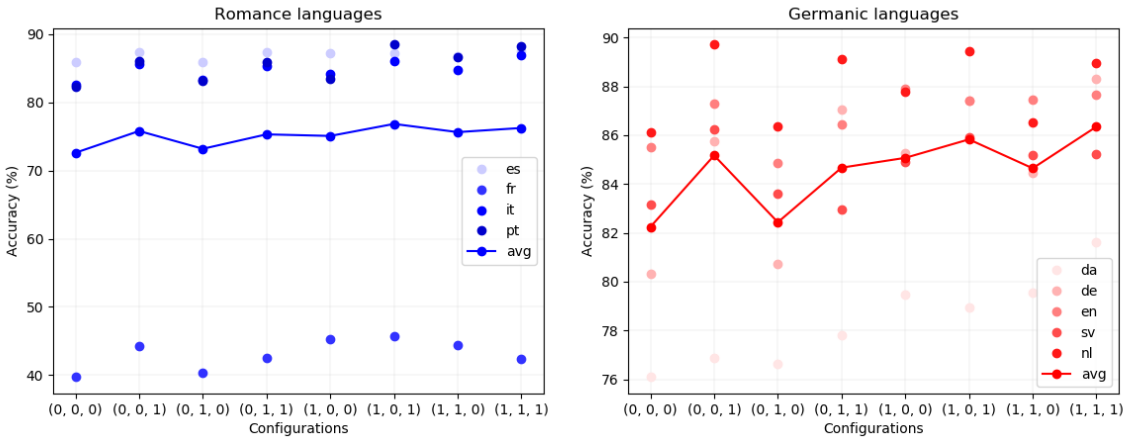
Table 4.3: The table of results for Blank Tag Method with Viterbi algorithm.

From the table 4.3 we see the best configuration was different from both best configurations from algorithm Random Tag and the algorithm Blank Tag with softmax function. Configuration (1, 1, 1) (word embeddings, character embeddings and auxiliary task included) gave the best results. Although the most stable version (since the maximum difference between results of each run has dropped on the 26.03%) it did not give the best results for all the

languages.

4.3 Results per Families

The results for each family are displayed on the following figures. We calculated the maximum results for each configuration in both algorithms and calculated the average results per each language family.

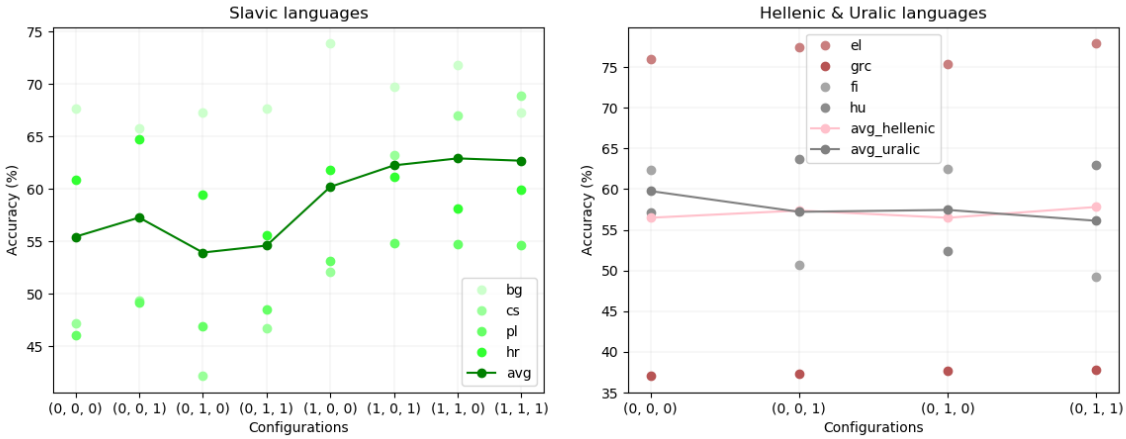


(a) Final results for Romance family.

(b) Final results for Germanic family.

Figure 4.2: From (a) we can see that Spanish was the most and French the worst accurately tagged from Romance family. Although the average accuracy is good (72-78%), the French language really lowers it down. On (b) we can see Dutch is the best and Danish is the worst tagged language from the Germanic family. The average accuracy is the highest out off all families (83-87%).

From Figures 4.2, 4.3 and 4.4 we notice that for Germanic languages we have achieved the highest accuracy compared to other families. The average accuracy exceeds 82% for each configuration. The Dutch language reached our total best accuracy (89.71%) for the algorithm Blank Tag using the Viterbi algorithm for a configuration (0, 0, 1), meaning only word embeddings were included. The Romanian languages were also very accurately tagged, except for French which had the maximum accuracy of only 44,38%. This is interesting finding since the Romance languages are known to be the most used language family in NLP researches. We looked at type coverage (9.5%) and token coverage (7.65%) for French, and although are not as low as expected, we compared them to our finest tagged language, Dutch which has a type coverage of 38.5% and token coverage of 69%. We also looked at the dictionary data: French added up to 17 863 tokens while Dutch almost four

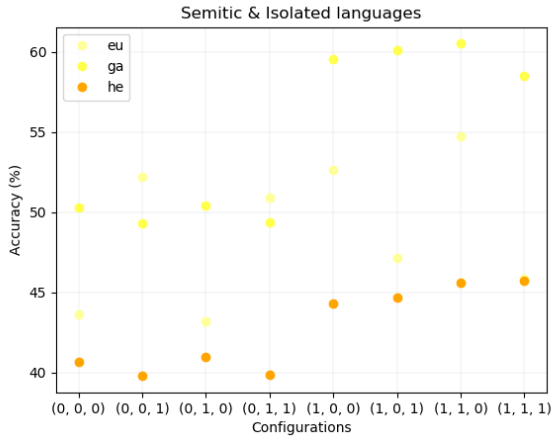


(a) Final results for Slavic family.

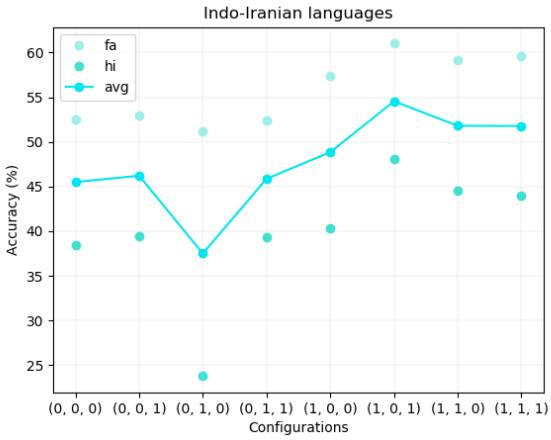
(b) Final results for Hellenic and Uralic families.

Figure 4.3: (a) Bulgarian was the best and Czech the worst accurately tagged from Slavic family. Although the average accuracy remained good (54-63%), the differences between the best and the worst accuracy were more than 25%((0,1,0) configuration). On (b) are results for Hellenic and Uralic languages for the first four configurations (absence of Polyglot word embeddings). There are only two languages per each family, but we can see that although Hellenic languages have big and Uralic languages have small accuracy difference, the average accuracies are pretty similar for all of the configurations.

times as much. Training files were also checked and resulted in Dutch having the train file two times bigger than French.
The lowest maximum accuracy of 37.74% was achieved for Ancient Greek language (Hellenic family) which had type coverage of only 2.64% and token coverage of 9.43%.
We also established that if the token coverage was higher than type coverage, the results were slightly lower than expected by Figure 4.1. This scenario usually appeared for languages with a small number of tokens in train files.



(a) Final results for Semitic language and two isolated language.



(b) Final results for Indo-Iranian family.

Figure 4.4: (a) are the final results for Hebrew (our only Semitic language) plus Basque and Irish (both language isolate so the results are not comparable). (b) presents Indo-Iraninan languages which have the lowest average accuracy out off all out off families (36-52%).

4.4 Related work

We compared our results to two different papers:

1. Supervised learning (Li, 2012.)
2. Unsupervised learning (Agić, 2015.)

Unsupervised and supervised methods use different setups, which greatly increased the degrees of freedom of the model allowing it to capture more fine-grained distinctions. The best-supervised system we are aware of that evaluate the Wiktionary is definitely (Li, 2012.). They presented four models (HMM, SHMM, HMM-ME, and SHMM-ME) for supervised learning which lie upon first and second order Hidden Markov Models plus feature-based max-ent emission. Agić also presented four models but for unsupervised learning, only these models rely on projection methods for parallel corpora.
We note that the results are not directly comparable since using a diverse data, the comparison with the first research is more accurate since using the Wikitionary dictionaries for our nine languages.

ISO	Random	Blank Tag & Softmax	Blank Tag & VIterbi	Supervised	Unsupervised
bg	61.88	**73.87**	72.51		
cs	49.45	**68.92**	68.83		
da	76.36	~~**82.08**~~	81.61	**83.3**	78.6
de	81.71	85.57	**88.29**	85.8	80.5
el	66.26	72.96	~~**77.87**~~	**79.2**	59.0
en	86.94	**88.29**	87.91	87.1	72.4
es	83.83	**87.50**	86.78	86.4	82.6
eu	47.21	**54.75**	52.63		
fa	52.94	**61.00**	59.14		
fi	**62.89**	59.89	62.44		
fr	35.85	44.38	**45.67**		
ga	49.42	**60.53**	60.11		
grc	35.90	37.10	**37.74**		
he	33.21	**45.74**	45.14		
hi	38.74	**48.10**	44.50		
hr	58.35	~~**64.79**~~	62.13		**67.8**
hu	49.48	63.00	**63.70**		
it	84.74	85.73	**86.85**	86.5	76.5
la	39.10	41.65	**44.78**		
nl	84.07	89.66	**89.71**	86.3	
pl	48.79	54.86	**56.06**		
pt	76.89	79.94	**88.46**	84.5	54.5
ro	74.06	75.72	**76.20**		
sv	86.24	**88.73**	84.72	86.1	74.7

Table 4.4: Maximum results compared to supervised and unsupervised methods.

From Table 4.4 we see improved accuracy for almost all languages (except for Danish and Greek). When comparing to the unsupervised learning, we improved accuracy for all except for Croatian (although the results are not fully comparable due different data).

Out of all of our models, bi-LSTM using algorithm Blank Tag with softmax function and (1, 0, 1) configuration had the most maximum scores out of 24 languages. It achieved maximum accuracies for Danish, English, Persian, Hindi and Swedish. We should mention, that this model includes word embeddings, which we not provided for Greek, Ancient Greek, Hungarian, Latin and Romanian, so it was not tested for them.

Chapter 5

Conclusion

We extent an (Plank, 2016.) approach to learning POS taggers with simply using Wiktionary dictionaries. We designed multiple bi-LSTM models which perform well across the 24 languages. The bi-LSTM tagger without lower-level bi-LSTM for subtokens falls short, although outperforms the on four languages (Irish, Finnish, Ancient Greek and Polish), hence we conclude bi-LSTM model clearly benefits from character representations. These stands for all $(0, a, w)$, where $a, w \in \{0, 1\}$ configurations. The combined configuration with word and character embeddings included ($(1, 0, 1)$ configurations) gave in average the best representation, outperforming the baseline on all except for German and Italian. By creating semi-supervised learning technique, we enabled ourselves to compare our results to both supervised and unsupervised methods. The methods outlined in the paper are standard and easy to replicate, yet highly accurate and should serve as baselines for more complex proposals. These encouraging results show that using free, collaborative NLP resources can in fact produce results of the same level or better than using expensive annotations for many languages. It would be very interesting and to arise across an even larger number of language types, especially non-European languages.

Bibliography

[1] Zeljko Agić, Dirk Hovy, Anders Søgaard. *If all you have is a bit of the Bible: Learning POS taggers for truly low-resource languages*. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing, pages 268-272, Beijing, China, 2015.
http://aclweb.org/anthology/P/P15/P15-2044.pdf

[2] D. Das, S. Petrov. Unsupervised Part{of{Speech Tagging with Bilingual Graph-Based Projections
http://aclweb.org/anthology/P/P11/P11\OT1\textendash1061.pdf

[3] D. Garrette, J. Mielens, J. Baldridge. *Real–World Semi–Supervised Learning of POS–Taggers for Low–Resource Languages*
http://aclweb.org/anthology/P/P13/P13\OT1\textendash1057.pdf

[4] Yoav Goldberg. *Neural Network Methods in Natural Language Processing* Bar-Ilan University, Synthesis lectures on human language technologies, MorganClaypool publishers, 2017.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning* MIT Press, 2016.

[6] Shen Li, Joao V. Graca, Ben Taskar. *Wiki-ly Supervised Part-of-Speech Tagging*. Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pages 1389–1398, Jeju Island, Korea, 2012.
http://aclweb.org/anthology/D/D12/D12-1127.pdf

[7] List of Names of Languages represented by ISO 639-2 standard codes.
https://www.loc.gov/standards/iso639-2/php/code_list.php

[8] Christopher Olah. Neural Networks, Types, and Functional Programming Blog post, 2015.
http://colah.github.io/posts/2015-09-NN-Types-FP/

[9] Christopher Olah. Understanding LSTM Networks Blog post, 2015.
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[10] Barbara Plank, Anders Søgaard, Yoav Goldberg. *Multilingual Part-of-Speech Tagging with Bidirectional Long Short-Term Memory Models and Auxiliary Loss*. Article, Cornell University Library, 2016.
https://arxiv.org/abs/1604.05529

[11] University College London (UCL) online lectures on Internet grammar.
http://www.ucl.ac.uk/internet-grammar/wordclas/open.htm

[12] Wikipedia article on language families
https://en.wikipedia.org/wiki/List_of_language_families

[13] D. Yarowsky, G. Ngai. *Inducing Multilingual POS Taggers and NP Bracketers via Robust Projection across Aligned Corpora*
http://aclweb.org/anthology/N/N01/N01\OT1\textendash1026.pdf

Sažetak

U radu je iznesena i objašnjena jednostavna metoda za učenje morfološkog označivaća za nisko resursne jezike oslanjajući se na rječnike tih jezika. Unatoč znatnom broju nedavno objavljenih radova koji oslovljavaju ovaj problem, bez nadzorne metode učenja nisu rezultirale dovoljno velikom točnošću. Jedna od metoda (slabo) nadzirnog učenja je korištenje paralelnog teksta između jezika s bogatih i siromašnim resursima koji znatno poboljšava točnost morfološkog označivanja. Međutim, paralelni tekstovi nisu uvijek dostupni, a tehnike za upotrebu istog zahtijevaju mnogo složenih algoritamskih koraka. U radu smo pokazali kako izgraditi jednostavan morfološki označivać pomoću bi-LSTM neuronskih mreža i slobodno dostupnog i prirodno rastućeg resursa, Wiktionary-a Za devet jezika koje smo označili podatke u svrhu procjene dobivenih rezultata, postižemo točnost koja u nekim slučajevima prelazi sve metode bez nadzora i metode s nadzorom koje koriste skrivene markovljeve lance i paralene korpuse.

Summary

We present a simple method for learning part-of-speech taggers for low-resource languages using dictionaries as are reference method. Despite significant recent work, purely unsupervised techniques for part-of-speech (POS) tagging have not achieved useful accuracies required by many language processing tasks. Use of parallel text between resource-rich and resource-poor languages is one source of weak supervision that significantly improves accuracy. However, parallel text is not always available and techniques for using it require multiple complex algorithmic steps. We have shown that we can build POS-tagger by using bi-LSTMs and a freely available and naturally growing resource, the Wiktionary. Across nine languages for which we have labeled data to evaluate results, we achieve accuracy that in some cases exceeds all unsupervised methods, supervised method which uses hidden Markov chaines and parallel text methods. We achieve highest accuracy reported for several languages.

Curriculum Vitae (CV)

I was born in 1993. in Split where I attended my primary school after which I attended a mathematical high school. In 2011., I started studying undergraduate programme of Mathematics on Faculty of Science at the University of Zagreb which I finished in 2015. After getting my Bachelor degree, I started the Master's programme of Computer Science and Mathematics on the same faculty. During my first year, I received a Rector's award for a student paper on topic "Mathematics in Politics". In the same year, I went to exchange student program on Faculty of Computer Science at University of Ljubljana, Slovenia. During the last three years of my education, I worked as a teaching fellow on the University of Zagreb. Last four months of my education, I spent on the internship at IT University of Copenhagen, Denmark during which this Master's thesis was written.